UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Desarrollo de una aplicación para la programación de un servicio de anestesia

Bruno Javier Blasco Smaranda
Tutor: Alfonso Ortega de La Puente

Julio 2017

Desarrollo de una aplicación para la programación de un servicio de anestesia

AUTOR: Bruno Javier Blasco Smaranda
TUTOR: Alfonso Ortega de La Puente

Grupo de la EPS (opcional)
Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio de 2017

Resumen (castellano)

El reparto de tareas entre trabajadores se ha realizado tradicionalmente a mano por diversos responsables. Esto es un trabajo tedioso que cuando muchas variables y restricciones puede llevar a errores y repartos poco óptimos que afectan seriamente al rendimiento de la plantilla.

Estudiado como *The Nursery Roster Problem* desde hace más de medio siglo, el problema se ha intentado abordar con numerosos algoritmos con el fin de obtener repartos óptimos teniendo en cuenta las preferencias de los trabajadores.

Este Trabajo de Fin de Grado trata de demostrar que no siempre es necesario aplicar técnicas complejas de inteligencia artificial para encontrar una solución y que en ocasiones es preferible delimitar el problema de tal modo que se ajuste a modelos matemáticos óptimos.

En este proyecto se implementa el algoritmo *Kuhn-Munkres* para el reparto de tareas de un turno de anestesia sobre una aplicación web completamente responsiva buscando rapidez y rendimiento mediante el uso de las nuevas tecnologías propuestas por el *W3C* en vez de pesados *frameworks front-end.*

La aplicación completamente automatizada permitirá a los trabajadores mostrar sus preferencias y realizar intercambios y al administrador tener control absoluto para modificar cualquier decisión tomada por el sistema.

Abstract (English)

Task scheduling among workers has been traditionally been done manually by personnel fit for such task. This is a tedious work which, when involving multiple variables and restrictions, often ends up opening up a road for mistakes and poor planning which may lead to serious performance issues among the workers.

This has been studied for more than half a century as *The Nursery Roster Problem*, approached with a large amount of different algorithms seeking to fulfil the preferences of the workers with an optimal roster.

This Bachelor Thesis seeks to prove how it is not necessary to apply extensively complex *AI* algorithms to reach an optimal solution and how instead it's often preferred to confine the problem properly so that it fits the solutions offered by mathematical algorithms.

This project implements the *Kuhn-Munkres* algorithm to schedule the anesthesiologist's shifts on a completely responsive web application which seeks speed and performance through the use of the newest technologies proposed by the *W3C* instead of heavy front-end frameworks.

The completely automatized application will enable the workers to input their preferences and perform task exchanges among themselves; whilst the granting the administration absolute control to override any decision made by the system.

Palabras clave (castellano)

Inteligencia Artificial, Problema de la Asignación, Nurery Rostering Problem, Planificación, NP, NP-Hard, Kunh-Munkres, Algoritmo Húngaro, Django, Framework, Anestesia, Turno, CSS3, Flexbox, Diseño Responsivo, Aplicación Web.

Keywords (inglés)

Artificial Intelligence, Assignment Problem, Nursery Rostering Problem, Schedule, NP, NP-hard, Kuhn-Munkres, Hungarian Algorithm, Django, Framework, Anesthesia, Shift, CSS3, Flexbox, Responsive Design, Web Application,

Agradecimientos

En primer lugar a mi familia, por encaminarme a estudiar esta carrera. A mi tutor, Alfonso Ortega, por empujarme en momentos de dificultad. A todos aquellos profesores que se han esforzado de verdad por el aprendizaje de sus alumnos. A mis amigos, por el soporte y a mi perro, por estar siempre ahí haciéndome compañía. Gracias.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 Motivación

Es muy común encontrarse en los trabajos realizados por una plantilla con turnos con que el reparto se realice de manera manual por un responsable. Esto en un principio puede parecer adecuado ya que una persona con experiencia puede realizar un reparto excelente de los recursos y a su misma vez funcionar como portavoz al que dirigirse si alguna vez quiere obtener información o realizar algún cambio.

A pesar de esto, existen problemas fundamentales en este sistema:

- Es un trabajo tedioso de realizar.
- Es muy difícil tener en cuenta todas las preferencias de los trabajadores.
- El que reparte puede carecer de la diligencia y motivación de repartir correctamente.
- El que reparte puede anteponer sus intereses personales al repartir.
- La planificación suele exponerse como una hoja pinchada sobre un tablón escrito a mano. Un trabajador puede despistarse y desconocer su planificación. Si no se encuentra en el recinto, la única opción para obtener la información es contactar con sus compañeros, que pueden desconocer su rol.
- Si dos trabajadores quieren realizar un intercambio, tienen que hablar entre sí y además tramitarlo con la persona que lleve el plan.
- El plan suele marcar únicamente el presente, por lo que es muy difícil tener en cuenta quién hizo qué en el pasado o saber qué va a asignarse en el futuro.
- Para indicar sus preferencias, los trabajadores tienen que hacerlo siempre a través del encargado de repartir. Existe también la posibilidad de que ciertos trabajadores hablen en nombre de otros, lo cual puede acarrear una fuerte confusión.
- Si la plantilla y las restricciones crecen mucho, la solución del problema se vuelve algo inviable que tendrá que solventarse de forma arbitraria.

Este proyecto busca subsanar estos problemas informatizando el proceso. Este es uno de los casos en los que la máquina suele hacer mucho mejor el trabajo.

Además, el tema del reparto de trabajos es algo que ha inquietado a la comunidad científica durante mucho tiempo.

Adicionalmente, nos encontramos en un punto bastante inestable dentro del mundo del diseño web, con la aparición de nuevas herramientas para solventar carencias graves y nuevas iteraciones de estándares revolucionarios para encajar el diseño web en plataformas móviles.

1.2 Objetivos

El objetivo básico es la realización de una plataforma web que reparta trabajadores entre distintos trabajos teniendo en cuenta restricciones y preferencias. Sin embargo, la realización del proyecto puede ser una herramienta para el aprendizaje, por lo que se puede encaminar para un beneficio personal.

En parte, los objetivos son los siguientes:

- Estudiar el estado del arte de el *Nursery Rostering Problem* y aplicar una de las soluciones existentes.
- Aprendizaje de un nuevo lenguaje de programación (en este caso, Python).
- Aprendizaje de un framework MVC completo (en este caso, Django).
- Aprendizaje del despliegue de un servidor en linux y el manejo con systemd.
- Exploración de las nuevas tecnologías web.
- Aprendizaje de Flexbox e implementación desde cero de una interfaz web completa con aspecto pulido, minimizando el uso de *hacks* y otras trampas como plantillas.

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- **Capítulo 1: Introducción.** Expone la motivación del proyecto, sus objetivos y la organización de esta memoria.
- **Capítulo 2: Estado del arte.** Formula el problema y los distintos algoritmos que se han empleado a lo largo del tiempo para solucionarlo, además como las distintas tecnologías de actualidad para el despliegue de una plataforma web.
- **Capítulo 3: Análisis.** Señala los requisitos que ha de cumplir el proyecto.
- **Capítulo 4: Diseño.** Explica la arquitectura de la aplicación y cómo funcionan las diferentes partes.
- **Capítulo 5: Desarrollo.** Entra en detalle en aspectos más técnicos del funcionamiento de la aplicación, haciendo hincapié en los problemas surgidos y cómo han sido resueltos.
- **Capítulo 6: Integración y pruebas.** Especifica la metodología seguida para integrar las distintas partes del código y la implantación de pruebas.
- **Capítulo 7: Conclusiones y trabajo futuro.** Realiza observaciones sobre los resultados obtenidos y herramientas utilizadas, así como la rutas rutas por las que puede progresar el proyecto.

2 Estado del arte

2.1 Estado del arte hospitalario [1, 2, 3]

2.1.1 Contexto histórico de la anestesia

El 16 de octubre de 1846 se realiza en el Hospital General de Massachusetts la primera demostración pública del empleo de éter por el dentista de Boston William T. G. Morton para anestesiar a un paciente con la motivación de extirpar un tumor bajo su mandíbula. Siete años más tarde, el anestesiólogo John Snow administró cloroformo a la Reina Victoria para asistirle en un parto sin dolor. Es entonces cuando se generaliza una aceptación social por el uso de anestésicos.

En un principio el trabajo del anestesiólogo es realizado por los propios cirujanos en el quirófano, pero con el transcurso del tiempo y debido a la complejidad de las tareas, la anestesia se delega como ocupación única a enfermeros o cirujanos hasta que nace la especialidad.

2.1.2 El servicio de anestesia

Si bien la función del anestesiólogo consistía únicamente en administrar anestésico, su rol ha evolucionado a lo largo del tiempo para abarcar competencias que se extienden más allá incluso del ámbito quirúrgico; incluyendo el mantenimiento de funciones vitales, el establecimiento de cuidados médicos para pacientes críticos, tratamiento del dolor, aplicación de métodos inherentes de la especialidad, formación del personal médico y paramédico en situaciones críticas, investigación y administración hospitalaria.
Es más, en la actualidad la figura del anestesista se ha convertido en la pieza clave para el funcionamiento quirúrgico. Es el anestesista quien da el visto bueno al paciente para operarse y el que toma el control del quirófano durante la cirugía. Esto es de relevante interés ya que decidir las rotaciones de los servicios de anestesia conlleva la base de la gestión hospitalaria.

2.1.3 Enfoque al reparto mediante software de turnos de anestesia en hospitales

En 2016 la clínica universitaria de Heidelberg [4] realizó un estudio consistente en un cuestionario telefónico a 35 hospitales universitarios con 20 preguntas referente a cómo se planifican los servicios y 15 preguntas adicionales por conocer el grado de satisfacción.
El estudio trata de valorar dos grupos: aquellos que manejan una aplicación para el reparto de turnos (bien sea diseñada por una empresa independiente o por el propio hospital) y aquellos que realizan la asignación de manera manual, incluyendo en este grupo el uso de aplicaciones de hojas de cálculo o similar.
De los 35 hospitales, 23 respondieron a la encuesta, promediando 105 médicos por hospital. En un 53% de los casos se emplea software (12 de 23), de los cuales 5 utilizan software comercial. El uso de software de reparto mejoró la eficiencia del reparto a corto plazo, sin mejora significativa a medio-largo plazo, además de disminución de los conflictos y mejora de la eficiencia de trabajo según el personal sanitario.
Hasta ahora se han empleado diversas plataformas para la organización de los servicios de anestesia, pero ninguna ha llegado a adoptarse como estándar global.

2.2 The nursery rostering problem

El "Nursery Rostering Problem", también conocido como "Nurse Scheduling Problem" es la tarea consistente en planificar los distintos turnos de trabajo del servicio de enfermería de un hospital, teniendo en cuenta los distintos intereses que puedan tener los enfermeros, como pueda ser trabajar en un rol determinado y de cumplir una serie de requisitos y restricciones impuestos por el propio hospital.
Convencionalmente, cada enfermero trabaja tres turnos diariamente: mañana, tarde y noche.
Se pueden considerar dos tipos de restricciones:

- **Restricciones fuertes:** Son aquellas que de no ser cumplidas, la planificación realizada para el servicio pasa a ser inválida. Un ejemplo de restricción fuerte es que un enfermero descanse seguidamente de haber realizado un turno de guardia, o que un rol que requiera un cierto grado de especialización y experiencia no sea asignado a un enfermero novel.
- **Restricciones débiles:** Son aquellas restricciones cuyo incumplimiento no supone una planificación inválida. Un enfermero puede mostrar su preferencia por la realización de una determinada tarea, o de evitar trabajar junto a cierto personal.

Así pues, la planificación de un servicio de enfermería tiene como objetivo cumplir todas las restricciones fuertes, tratando de satisfacer el máximo de las restricciones débiles.
Este problema es de especial relevancia para la comunidad científica y ha sido estudiado durante más de 40 años y catalogado como un problema de complejidad *NP-hard*. Esto se debe a la alta variabilidad de requisitos y restricciones entre problemas. Los términos "Nursery rostering" y "Nursery scheduling" no se limitan únicamente al reparto de turnos en enfermería, sino que se entiende como el problema generalizado de reparto de turnos para cualquier trabajo.

2.2.1 Algoritmos para la resolución del problema

Burke et al. (2004) recopila los diferentes algoritmos empleados para la resolución del problema:

- **Modelos de optimización matemática:** La programación matemática es apropiada para la obtención de soluciones óptimas. No obstante, su limitación fundamental es la complejidad y tamaño del espacio de búsqueda. La mayoría de investigadores acotan el problema a un subconjunto de restricciones. La mayoría de experimentos se fundamentan en la acotación de una única función objetivo, aunque una porción de ellos también se han desenvuelto con programación multi-objetivo.
- **Modelos basados en la programación declarativa:** Estos modelos emplean motores de inferencia como prolog para la resolución de restricciones como cláusulas.
- **Uso de sistemas expertos:** Basados en reglas if-then, permiten soporte interactivo para el desarrollo de la planificación.
- **Modelos basados en heurísticas:** Requieren una especificación muy clara de la calidad de distintos repartos. Mayoritariamente desarrolladas para producir ciclos emulando prueba y error.
- **Modelos basados en metaheurísticas:** Los más efectivos cuando el nivel de restricciones es muy elevado y encontrar una solución posible es muy difícil. Estos modelos están eclosionando en la actualidad: enfriamiento simulado, búsqueda tabú, algoritmos genéticos, algoritmo de colonia de hormigas...

2.3 El algoritmo húngaro [5,6]

El nombre "algoritmo húngaro" surge del mérito de los matemáticos Dénes Kőnig y Jenő Egerváry, sobre los que el matemático estadounidense Harold W. Kuhn basó su trabajo publicado en 1955 en el cual resuelve el problema de la asignación en el contexto de reparto de trabajos a trabajadores. En el documento detalla el problema, los fundamentos matemáticos necesarios para el funcionamiento del algoritmo para finalmente explicar el algoritmo con ejemplos.

2.3.1 Definición del problema de la asignación

Partamos del hipotético escenario en el que Juan, Paco, Ana y María son compañeros en un proyecto de la Escuela Politécnica. Deciden dividir el trabajo en cuatro secciones para después repartirlo. Desafortunadamente, cada parte del proyecto se ha de completar de manera secuencial. Como es natural, cada integrante del equipo posee cualidades distintas que les vuelve más competentes en realizar unas tareas con mayor facilidad que otras.
Los cuatro integrantes del grupo deciden en conjunto el tiempo en minutos en que estiman tardar en completar cada tarea:

	Juan	Paco	Ana	María
Interfaz	83	36	34	98
Algoritmo	42	75	39	5
Red	47	42	52	34
Sonido	10	35	28	72

Tabla 1: Posible solución del problema de la asignación

El objetivo del reparto consiste en minimizar el tiempo requerido para completar el proyecto.
Una posible solución sería el reparto de las casillas marcadas en verde. En tal caso supondría un tiempo total de 10+36+39+34=119.

2.3.2 Rendimiento

Un enfoque ingenuo en resolver el problema consistiría en probar todas las combinaciones posibles de reparto y contrastar los costes totales obtenidos en cada caso. Siendo 'n' el tamaño de la tabla, calcular todas las permutaciones posibles supondría un rendimiento de O(n!). Esto puede ser admisible para un tamaño de tabla muy reducido, pero para repartos perfectamente naturales de 20x20, el coste de cómputo se vuelve completamente inviable.

James Munkres, profesor emérito en matemáticas de MIT demostró que el rendimiento del algoritmo húngaro es fuertemente polinómico. Desde entonces el algoritmo se conoce también bajo el nombre de Kuhn-Munkres.
Originariamente el algoritmo tenía un rendimiento de $O(n^4)$ pero en base a mejoras se puede obtener un rendimiento de $O(n^3)$.

Véase el anexo para la explicación del algoritmo

2.4 Bases de datos

La base de datos va a ser la responsable de almacenar la información persistente de la aplicación. En este apartado se van a evaluar las diferentes opciones de gestores de bases de datos considerados como los principales.

330 systems in ranking, June 2017

Rank			DBMS	Database Model	Score		
Jun 2017	May 2017	Jun 2016			Jun 2017	May 2017	Jun 2016
1.	1.	1.	Oracle	Relational DBMS	1351.76	-2.55	-97.49
2.	2.	2.	MySQL	Relational DBMS	1345.31	+5.28	-24.83
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1198.97	-14.84	+33.16
4.	4.	↑ 5.	PostgreSQL	Relational DBMS	368.54	+2.63	+61.94
5.	5.	↓ 4.	MongoDB	Document store	335.00	+3.42	+20.38
6.	6.	6.	DB2	Relational DBMS	187.50	-1.34	-1.07
7.	7.	↑ 8.	Microsoft Access	Relational DBMS	126.55	-3.33	+0.32
8.	8.	↓ 7.	Cassandra	Wide column store	124.12	+1.01	-7.00
9.	9.	↑ 10.	Redis	Key-value store	118.89	+1.44	+14.39
10.	10.	↓ 9.	SQLite	Relational DBMS	116.71	+0.64	+9.92

Figura 1: Ranking de gestores de bases de datos de acorde a db-engines.com

2.4.1 RDBMS

Basado en el trabajo de E. Codd e implementando álgebra relacional, son un modelo de base de datos muy extendido. Los gestores de bases de datos relacionales almacenan en un servidor la información a modo de tablas normalizadas, donde el dato está restringido a ciertos tipos. La base de datos queda separada de la aplicación que la emplee y para la interacción con la misma se utiliza el lenguaje de query estructurado (SQL).
Ante un crecimiento del volumen de datos, el escalado es vertical, precisa de la mejora del servidor en el que se hospede el gestor.

Por norma general, las transacciones satisfacen las propiedades ***ACID** (en inglés)*:

- **Atomicidad:** La transacción sólo se lleva a cabo en su completitud. Si la transacción falla en algún punto, no se verá reflejado ningún cambio en la base de datos. Esto debe de ser así incluso en casos como pueda ser un apagón.

- **Consistencia:** Para poder realizarse la transacción, se ha de comprobar que se mantenga la integridad de los datos en base a reglas, como puede ser borrado en cascada o restricciones de duplicidad.

- **Aislamiento:** Impide condiciones de carrera entre distintas transacciones. Una aplicación concurrente debe tener el mismo efecto que una ejecución secuencial.

- **Persistencia:** Una vez realizada una transacción, los cambios han de perdurar. En otras palabras, si hubiese un apagón tras realizar la transacción, los cambios no pueden perderse.

2.4.2 Gestores RDBMS [12]

- ***Oracle SQL:*** Definido como ORDBMS, integra orientación a objetos en la base de datos. Accesible desde un número muy elevado de lenguajes de programación. Siendo uno de los productos estrella de Oracle, parece el más indicado para la integración con JavaEE. Licencia propietaria.

- ***Microsoft SQL:*** Enfocado a las soluciones empresariales de Microsoft. Se encontraba disponible únicamente para Windows (en 2016 se anunció soporte a Linux) y la licencia es propietaria. Posee un motor más lento y pesado y una suite de herramientas analíticas.

- ***MySQL:*** Componente fundamental en LAMP junto a PHP y apache. Fue adquirido por Oracle en 2010. Es ligero y de código abierto. Al ser muy popular, posee una comunidad muy grande por lo que es muy fácil resolver problemas relacionados con MySQL.
  El rendimiento es algo inferior, aunque se puede mejorar sacrificando ACID y es conocido por no adherirse completamente al estándar SQL.
  Su licencia es GPL2, aunque precisa de licencia comercial para funcionalidad adicional.

- ***PostgreSQL:*** ORDBMS, permite de forma nativa la orientación a objetos. En el momento tiene la implementación más completa de SQL que mejor se adhiere al estándar y su rendimiento es excelente. La licencia de este producto es liberalmente abierta.

- ***MariaDB:*** Surgió como *fork* de MySQL por un número de desarrolladores del mismo por las preocupaciones ocasionadas por la adquisición de Oracle. MariaDB busca mantener un alto grado de compatibilidad con MySQL.
  Al ser un *fork* reciente de MySQL, comparte sus ventajas e inconvenientes, siendo incapaz de operaciones como *intersect, except, merge joins, common table expressions o queries paralelas*. No obstante, a diferencia del anterior, sí soporta funciones de ventana.

- ***DB2:*** Desarrollado por IBM originariamente para sus plataformas, DB2 es robusto, rápido y con un grado de funcionalidad similar al de Oracle. Su licencia es comercial.

- ***SQLite:*** *Es la solución embebida:* SQLite no maneja un servicio al que conectarse como los demás, sino que funciona como una librería integrada en la aplicación que la emplea. La base de datos se puede almacenar como un mero fichero.
  Debido a su disponibilidad y facilidad de uso, es la plataforma SQL más desplegada del mundo.
  Su implementación SQL no es estándar y tiene ciertas carencias como no soportar RIGHT joins.

2.4.3 NoSQL [13]

Las bases de datos relacionales no son la única manera de almacenar información. Como alternativa existen las tecnologías conocidas como *no-sql* o *not-only-sql*.
NoSQL en realidad no es nada nuevo, su uso ya data desde 1960, aunque las bases de datos de este tipo no fuesen conocidas bajo este nombre.

El nombre NoSQL ha sido adoptado recientemente (en torno a 2010) y su uso se ha potenciado por grandes corporaciones como Google, Facebook y Amazon con la motivación de iteración al progreso con el denominado Web2.0.

La distribución de datos en NoSQL no se dispone en tablas normalizadas predeterminadas con un número de columnas fijo de tipo preestablecido, sino de otras formas más similares al manejo de diccionarios en python o construcción de objetos JSON. Esto tiene la gran ventaja de que los datos almacenados pueden construirse de forma dinámica al no estar restringidos en columnas.

Los modelos principales de almacenamiento de datos son *key-value, column-oriented (análogamente también existe row-oriented),* orientado al almacenamiento de documentos, orientado a grafos e incluso en algunos casos usando también álgebra relacional.

Debido a la forma en que se almacenan los datos, suelen ser más simples de utilizar y tienen la ventaja de no requerir conocimientos de bases de datos relacionales, por lo que se encuentran a disposición de cualquier tipo de programador.

A diferencia de los RDBMS convencionales, NoSQL tiene la gran ventaja de no encontrarse centralizado, sino que está enfocado a la implementación distribuida en múltiples servidores. Esto tiene la gran ventaja de que escala horizontalmente en vez de verticalmente. En otras palabras, para ampliar el volumen de la base de datos y el tráfico dirigido a la misma puede contratarse más máquinas en vez de tener que mejorar el servidor existente. Apple, por ejemplo, cuenta con más de 75000 nodos de Cassandra con más de 10 Petabytes de datos.

Como inconveniente principal, NoSQL no suele satisfacer *ACID* y no existe ningún estándar, usando cada gestor su propio lenguaje especializado

De los gestores de datos NoSQL merece la pena destacar los siguientes:

- ***MongoDB:*** Almacena los datos en formato *BSON*, una forma de JSON binario. Es sencillo de utilizar y lidera como base de datos de documentos.

- ***Cassandra:*** Posee un lenguaje denominado CQL, ideal para programadores experimentados en SQL. Escala muy bien con grandes volúmenes de datos y se puede ajustar el nivel de consistencia.

- ***Redis:*** La herramienta más popular de tipo (key, value), tiene un rendimiento excelente y se puede hospedar directamente en memoria.

- ***HBase:*** Parte de Hadoop, reside sobre HDFS y está enfocado a Big Data.

2.5 Tecnologías disponibles para la lógica de la aplicación

2.5.1 Aplicaciones nativas

Teniendo en cuenta que el objetivo final de la aplicación es ser desplegado sobre un ordenador hospitalario, como mínimo se ha de poder utilizar desde Windows.

- **Win32 y COM:** Librerías elementales de windows para la creación de aplicaciones con GUI.

- **.NET**: Nueva generación de aplicaciones de Microsoft. Se puede desarrollar en **Windows Forms** o en el más nuevo **Windows Presentation Foundation.**
  Tiene la ventaja de poder ser extendido a *Windows Phone*.

También existen otras alternativas no tan exclusivas de Windows como **Tk**, Java **Swing** / **AWT**, o **Qt**.

En cuanto a dispositivos móviles la plataforma dominante es Android, aunque iOS también tiene un uso significativo. Cada plataforma tiene su propio framework con sus propios lenguajes, lo cual supone el sobrecoste de tener que realizar el cliente de aplicación tres veces para poder cubrirlas todas.

Existen también soluciones multiplataforma que permiten escribir la aplicación una única vez y desplegar tanto en Windows, Android, iOS como en plataformas adicionales.

- **Qt:** Proyecto muy maduro y de código abierto. Numerosas aplicaciones lo utilizan. Con **QtQuick** se puede escribir rápidamente aplicaciones con la interfaz definida claramente en ficheros **QML.**

- **WebApps:** Encapsulan la aplicación como una pequeña aplicación web que se ejecuta por un pequeño navegador integrado internamente. Existen numerosas alternativas para el desarrollo de estas aplicaciones, tales como **Apache Cordova**, **Gulp**, o **Atom**. Eso sí, el rendimiento es significativamente más bajo que con aplicaciones nativas, pero funcionará en la mayoría de dispositivos.

Independientemente del *framework* que se escoja para la aplicación del cliente, siempre será necesario un servidor centralizado que gestione el reparto (a no ser que se opte por una aplicación única).

2.5.2 Plataforma web

La plataforma web tiene la gran ventaja de ser desarrollada una única vez y poder ser accesible desde cualquier dispositivo que disponga de un navegador. No requiere de instalación de software en los dispositivos clientes.

Sigue la arquitectura *cliente-servidor*, donde se puede decidir dar más carga de trabajo al servidor o al terminal que accede. Además, puede protegerse en una red interna o ser accesible desde internet, abriendo paso a extenderse otras vías de acceso como APIs REST.

2.6 Back-end [11]

Back-end es la parte del servidor encargada de la lógica interna de la aplicación. Existe una gran variedad de lenguajes y frameworks en los que poder implementarse.
A continuación se listan las principales opciones valoradas para este proyecto:

2.6.1 Laravel

Es el *framework* MVC de PHP más popular con diferencia. Posee una documentación excelente, con una comunidad muy grande y un sistema de aprendizaje mediante videos denominado Laracasts. Maneja un gestor de paquetes denominado Composer. Emplea un ORM propio llamado Eloquent y el lenguaje de plantillas Blade.
La integración con otros *frameworks front-end* está preparada desde un principio.
La gran desventaja reside en que PHP es un lenguaje de programación lento.

2.6.2 ASP.NET

ASP surgió como la alternativa de Microsoft a PHP. En la actualidad Microsoft a unificado todas las tecnologías web de ASP.NET en lo que denomina MVC6.
Es un *framework* de tamaño corporativo con muy buena integración con el resto de herramientas de diseño que ofrece Microsoft; lo cual también es una gran desventaja al estar tan acoplado al resto del sistema.

2.6.3 Java

Java es un ecosistema muy extenso con buen rendimiento sobre el que se están desarrollando nuevos lenguajes como scala o incluso más recientemente Kotlin. Con JavaEE se pueden desarrollar aplicaciones web a nivel empresarial muy extensibles a gran escala. Entre sus frameworks destacan *Struts*, basado en sus tecnologías estándar (javabeans, servlets, jsp..) o Play, que busca simplicidad inspirado Rails.

2.6.4 Django

Framework MVC python. Su documentación es extensa, detallada y de buena calidad, con una comunidad muy grande. Es rápido, seguro y escalable cubriendo una funcionalidad muy extensa: ORM, internacionalización, sección de administración, sistema de autenticación…
Su motor de plantillas ha servido como inspiración para otros motores como Jinja.

2.6.5 Flask

Es un framework MVC python minimalista y modular. Está diseñado para integrarle otros componentes cuando se vea necesario, como pueda ser el ORM SQLAlchemy.

2.6.6 Rails

Similar en cuanto a funcionalidad a Django, pero escrito en Ruby. Es más conciso, menos explícito y con una comunidad algo más grande.

2.7 Front-end [11]

En el contexto de este proyecto definiremos *front-end* como la parte de la interfaz que será evaluada en el navegador para interactuar con el usuario.

Tradicionalmente las páginas web fueron creadas de tal modo que el contenido se mostrase de arriba a abajo como un documento, similar a como se ve estructurada esta memoria.

No obstante, comenzaron a surgir necesidades estéticas que requería ubicar elementos con una determinada disposición espacial.

Para ello, se usaban tablas para la estructuración del contenido, y poco más adelante *hacks* no muy limpios involucrando *float* y *clearfix*, consistente en abusar en la propiedad de flotación de imágenes en distintos contenidos, forzando que sean despejados lateralmente.

También existe el problema de la existencia de múltiples navegadores que interpretan las hojas de estilo en cascada de formas ligeramente distintas. Especialmente relevante en este caso es *Internet Explorer*, navegador más utilizado en el momento y que menos se adhería al estándar.

2.7.1 Flash

Flash fue creado en 1996 por Macromedia (después adquirido por Adobe) como un plugin para el navegador que superó con creces las limitaciones de diseño web del momento, y dotó de total libertad multimedia.

Flash. Como grave desventaja, es muy pesado requiriendo mucho cómputo y un vector principal de vulnerabilidades de seguridad.

Con el auge de las plataformas móviles y la aparición de HTML5 se encuentra en la actualidad en decadencia.

2.7.2 Frameworks [11]

En torno a 2007 se revoluciona el mercado con la aparición de las plataformas móviles.

Las páginas *responsivas* son aquellas que se redimensionan y amoldan a la pantalla del dispositivo y pueden mostrarse adecuadamente independientemente de si se trata de un móvil, una tableta o un ordenador.

Foundation y **Bootstrap** lideraron el cambio como frameworks CSS. Solucionaban el problema de CSS inconsistente entre navegadores creando una capa de abstracción por encima.

El diseño fundamental consiste en dividir la página en 12 columnas y dotar de un aspecto común a los diferentes elementos de la página.

Así se obtiene con facilidad una página con un buen aspecto sin la necesidad de tener que confeccionar el estilo por uno mismo. Desafortunadamente acarrea el problema de mezclar

información del estilo, una sobrecarga de componentes que probablemente no se vayan a utilizar y que el aspecto de las páginas diseñadas con el *framework* son idénticas entre sí.

Actualmente está creciendo el uso de *frameworks* javascript, que emplean javascript para confeccionar una pequeña aplicación en el cliente para intercambiar información con el servidor de manera concurrente. Entre ellos se encuentran **AngularJS, React, Vue.js, Ember** o **Meteor.**
A pesar de facilitar considerablemente el diseño de la página, supone sobrecargar al cliente con toda la maquinaria javascript, algo que se nota especialmente en dispositivos móviles. Una conexión lenta supondrá además que la página no cargue correctamente.

2.7.3 Nuevos estándares de W3C

2.7.3.1 Media queries [10]

Las media queries permiten aplicar reglas CSS condicionalmente si se cumplen ciertos criterios como el color, tamaño de la pantalla, el tipo de dispositivo y orientación

```
@media (min-width: 700px), handheld and (orientation: landscape) { ... }
```

Esto permite reestructurar la página sobre la marcha y volverla responsiva sin la necesidad de emplear javascript.
El inconveniente de las *media queries* es que se aplican a nivel de página, cuando darían mucha más flexibilidad si se pudiesen aplicar a nivel de elemento.

2.7.3.2 Flexbox [8]

Flexible Box Layout permite definir un elemento HTML como un contenedor en el cual los elementos se disponen en fila o en columna.
Se puede especificar la distribución y espacio, así como cómo se desenvuelven los elementos en su interior.

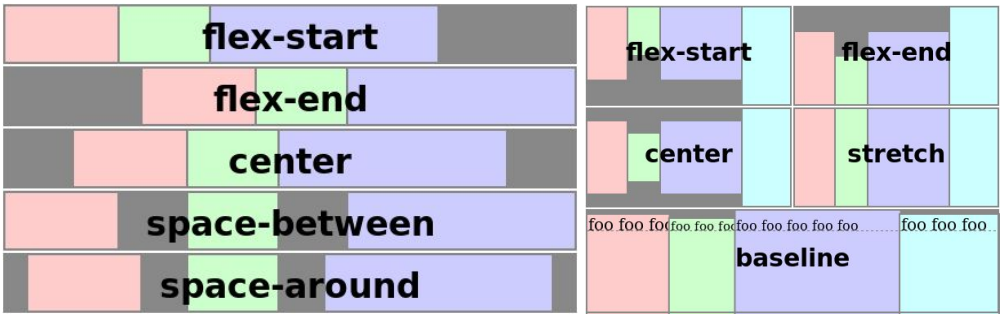


Figura 2: Distribución y alineación de contenido en Flexbox

Flexbox se encuentra ya en la etapa de CR (Candidate Recommendation) de w3c con una adopción de los navegadores de un 97%, por lo que se trata de una solución real y estándar para el *responsive design*.

2.7.3.3 Grid [9]

A diferencia de con Flexbox, donde el contenido dicta el layout; con Grid se opta por el enfoque de que sea el layout el que restringe el contenido.
Además, si Flexbox estaba diseñado para la distribución unidimensional del contenido en forma de fila o columna (pese a poder rebosar en múltiples), Grid está pensado para una distribución en dos dimensiones.

Figura 3: Ejemplo de layout flexbox.

Figura 4: Ejemplo de layout grid.

Grid permite definir de forma explícita dónde se posiciona cada uno de los elementos del HTML, lo cual permite que dicho fichero esté completamente libre de aspectos de diseño.

```
<div id="grid">
  <div id="title">Game Title</div>
  <div id="score">Score</div>
  <div id="stats">Stats</div>
  <div id="board">Board</div>
  <div id="controls">Controls</div>
</div>
```

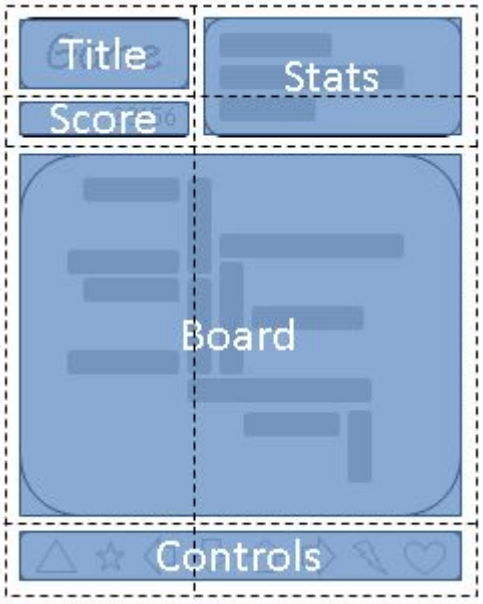


Figura 5: HTML claro para la estructuración con grid.

Grid parece ser la solución definitiva para el diseño de la organización de las páginas web. En la actualidad tiene un grado de adopción entre los navegadores del 70%.

3 Análisis de requisitos

En este apartado se estudian los diversos requisitos que ha de cumplir el sistema.
En este proyecto en particular es imperativo afianzar los requisitos por dos motivos fundamentales:

- Dada la alta variabilidad de funcionamiento en los contextos hospitalarios, es posible que aparezcan distintas formas de resolver el problema del reparto de turnos y los requisitos se vuelvan volátiles y contradictorios. Fijar los requisitos impide este problema.

- Dada la especificidad y coste de implementación del algoritmo, la realización de cambios puede resultar desastrosa al poder afectar a múltiples áreas del proyecto que precisarían rehacerse.

Los requisitos serán clasificados entre funcionales, que especificarán las pautas de funcionamiento de la aplicación para poder ser considerada completa; y no funcionales, que valorará aspectos de calidad del sistema como puedan ser la seguridad, estabilidad o usabilidad.

3.1 Requisitos funcionales

3.1.1 Funcionalidad general

- Funcionalidad para repartir un *pool* de trabajos entre un *pool* de trabajadores.
- Rutina para que el reparto ocurra de forma automática y recurrente.
- Adherencia a restricciones fuertes.
- Minimización de violación de restricciones débiles.
- Sólamente se debe poder acceder a la página por usuarios registrados.
- El reparto de tareas deberá ser homogéneo.
- Diariamente, cada trabajador o tarea sólamente podrán aparecer una única vez, es decir, una misma tarea no se realizará por dos trabajadores y un trabajador no realizará dos tareas.
- Entre los trabajos deberá existir la guardia: implica un turno de noche y el trabajador deberá estar libre el día siguiente.

3.1.2 Funcionalidad para el administrador

- Creación de usuarios.
- Definición de roles y grupos para usuarios para que puedan realizar tareas de administrador con limitaciones especificadas por el propio administrador.
- Creación de trabajadores.
- Permitir la asociación entre usuarios y trabajadores.
- Permitir la modificación de detalles de usuarios y trabajadores.
- Definición de trabajos.
- Permitir otorgar preferencia a ciertos trabajos sobre otros.
- Creación de **restricciones fuertes** que impidan a **trabajadores** trabajar en determinados trabajos.
- Definir planes para distintos intervalos de tiempo en los que se engloben los trabajos que se deben realizar y los trabajadores disponibles.
- Capacidad de realizar cambios manuales en las decisiones realizadas por la aplicación, incluso si conlleva violar restricciones. Esto incluye intercambios.

3.1.3 Funcionalidad para cualquier usuario

- Ocultar secciones de la página que no deban ser accesibles (por no ser administrador o por no tener trabajador asociado)
- Los trabajadores podrán definir su preferencia por ciertas tareas, y deberá ser respetada en la medida de lo posible.
- Poder cambiar su información personal. Posibilidad de emplear una imágen de perfil que aparezca en los repartos.
- Poder solicitar intercambios con otros trabajadores sin requerirse la intervención del administrador.
- Poder aceptar o denegar intercambios con otros trabajadores sin la necesidad de la intervención del administrador.
- En caso de intercambiar una guardia, se deberá intercambiar también el descanso.
- No se permitirá solicitar ni admitir un intercambio si cualquiera de los trabajadores es incompatible.
- Aceptar un intercambio implicará rechazar pendientes para el mismo trabajo.

- No se permitirá formular ni admitir intercambios que afecten en cascada a múltiples días.
- Restauración de contraseña en caso de olvido.
- Podrá ver listada la planificación por semanas.

3.2 Requisitos no funcionales

- La aplicación debe poderse instalar dentro de una red interna, accesible desde ordenadores comunes.
- Si el usuario abandona la aplicación durante más de 20 minutos, deberá caducar la sesión para que otros usuarios no actúen en su nombre.
- La aplicación debe de ser multiplataforma, cubriendo desde windows, linux y mac hasta smartphones. Esto incluye la necesidad de una interfaz adecuada para dispositivos móviles.
- La aplicación deberá tener una interfaz con un con un aspecto a la altura de los estándares de similares aplicaciones comerciales.
- La aplicación ha de ser ligera en el uso del cliente, considerándose ejemplos de aplicaciones pesadas (web o no) las de Facebook.
- El reparto de 20 trabajadores y 20 tareas en un día no ha de superar los 3 segundos.
- La aplicación deberá respetar **ACID**.
- La aplicación deberá ser libre y gratuita, sin ataduras de licencias propietarias.
- Una acción completa no deberá suponer más de 3 clicks.
- Todas las entradas de datos deberán ser saneadas. Por ejemplo, se deberá comprobar que un email contenga una arroba.
- Si se emplean metodologías web, la página debe estar protegida de *clickjacking*.
- La aplicación deberá poder funcionar adecuadamente incluso cuando el administrador haya hecho cambios inadecuados a la base de datos, como pueda ser haber violado una restricción fuerte

4 Diseño

En este apartado se detalla la arquitectura de la aplicación, así como del funcionamiento de los diferentes bloques de los que se compone y de las diferentes herramientas escogidas para soportarlo.

4.1 Modelo Vista Controlador

MVC es la arquitectura fundamental que emplea la aplicación. Desde la concepción de el patrón Modelo Vista Controlador, se han dado distintas interpretaciones del mismo, pero la idea generalizada consiste en una separación entre los datos, la representación y una capa que los comunica.

- **Modelo:** Alberga los datos de la BBDD, así como los métodos de acceso y modificación de la misma. Esta capa alberga también la lógica de dominio, como pueda ser el resultado de agregar múltiples tablas.
- **Vista:** Es la parte de la aplicación que muestra la información al usuario final. No contiene apenas lógica, lo único que hace es tomar resultados ya procesados y formateados para mostrarlos por pantalla.
- **Controlador:** Hace de pegamento entre la vista y el modelo. La interacción del usuario pasa por el controlador, solicita una petición al modelo (que realiza cambios en la base de datos y genera un resultado). Es el controlador el que toma el resultado y lo transfiere a una vista para mostrarlo por pantalla.
  El controlador contiene la lógica de la aplicación. En este caso, al ser grande, se ha separado en un módulo independiente.

4.2 ¿MVC o MTV?

Django se autodefine como "Model Template View" esgrimiendo como argumento que *View* es el contenido por mostrar mientras que *Template* es cómo se muestra el contenido. Argumenta entonces que el controlador es el propio *framework*.
A pesar de que lo que django define como *View* sólamente es una gran porción del controlador en el sentido estricto de la palabra, en este proyecto lo valoramos desde el sentido práctico de la estructuración del código. De tal modo:

Nombre según Django	Nombre usado en la práctica
Model	Model
Template	View
View	Controller

Tabla 2: Ranking de gestores de bases de datos de acorde a db-engines.com

4.3 Modelo

El modelo define la estructuración relacional que va a tener la base de datos. Se ha optado por emplear la notación de Chen al considerarse la notación más explícitas de las diferentes alternativas disponibles.

Para este proyecto en concreto, el modelo es ideal para comprender el problema en sí, ya que independientemente de la implementación todas las entidades tienen significado conceptual.

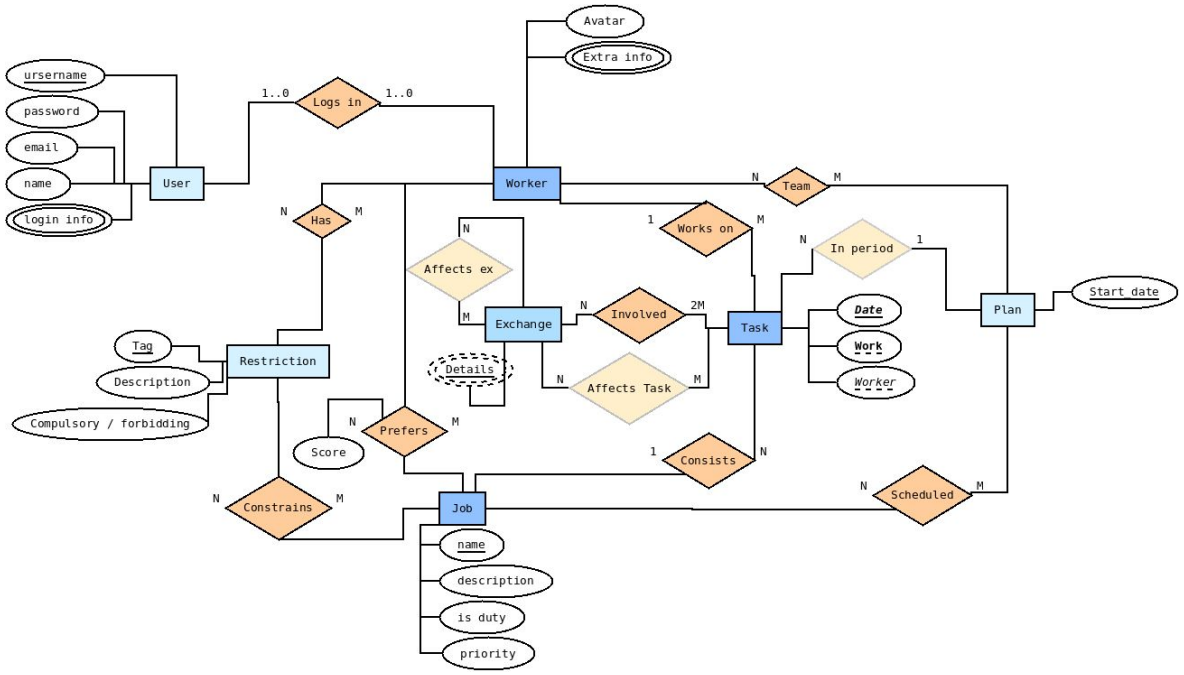


Figura 6: Diagrama entidad-relación

Las entidades marcadas con tonos más oscuros tienen mayor relevancia en cuanto a la funcionalidad básica que ha de cumplir el proyecto. Las relaciones aclaradas son derivadas.

4.3.1 Entidades y relaciones

4.3.1.1 User

El usuario desde el que se puede hacer login y manejar la aplicación. Puede tener (o no) asociado un trabajador. El administrador (o superusuario) tiene acceso al panel de administración y puede otorgar a otros usuarios la propiedad de "*staff*", que también permitirá el acceso al panel de administración restringido a determinados permisos asignados por el administrador.
Si el usuario no tiene un trabajador asociado, no tendrá acceso a su funcionalidad desde la plataforma.
Almacena cualquier información relevante al *login*, como un email a donde enviar la nueva contraseña si se pierde, desde cuándo es miembro o la última vez que realizó *login*.

4.3.1.2 Worker

Es la representación del anestesista dentro de la planificación. No hay que confundirlo con el usuario, ya que una instancia de trabajador puede aparecer dentro del reparto sin necesariamente contar con interacción con la aplicación. Guarda información para representarlo en el tablón semanal, incluyendo una imagen de perfil.
No obstante, si tiene un usuario asociado se derivará de ahí la información primero (como el nombre completo).

4.3.1.3 Job

Definición de un trabajo por realizar. Dentro del ámbito de anestesia es común que refiera la ubicación del trabajo, p. ej: Quirófano obstetricia, Consulta 2, Reanimación…
El trabajo **no es una instancia** temporal, sólamente define el qué.

- ***Name:*** Identificador del trabajo.
- ***Description:*** Pequeña descripción de en qué consiste el trabajo.
- ***Priority*** es una escala del 0-9 estableciendo la prioridad de que se realice la tarea. Si no fuese posible distribuir todas las tareas entre los trabajadores en un día debido a restricciones fuertes, los trabajos con una prioridad inferior se dejarán sin asignar para hacer espacio a los trabajos prioritarios.
- ***Duty*** marca el trabajo como una guardia, que se tratará de forma especial: Trabajo de guardia implica descanso al día siguiente.

4.3.1.4 Task

Es la asignación de un trabajador a una tarea en un día. Como restricción no puede existir en un mismo día dos veces el mismo trabajador o trabajo. No obstante, cualquiera de las dos relaciones foráneas pueden ser nulas, significando que el trabajo se ha marcado como "sin trabajador" o el trabajador ha sido marcado bien como descansando o como disponible auxiliarmente.
Si no existen tareas en un día, es porque todavía el planificador no ha realizado asignaciones.
Si en algún momento se realiza un intercambio, se deberá actualizar el cambio en la tabla de tareas.

- ***Worker:*** Trabajador asignado a la tarea. Puede ser nulo.
- ***Job:*** Trabajo en el que consiste la tarea. Puede ser nulo para representar libranza / comodín.

4.3.1.5 Exchange

Representa un intercambio de dos tareas entre dos trabajadores.

- ***Request time:*** Cuándo se ha solicitado el intercambio.
- ***Requester:*** El trabajador que ha solicitado el intercambio.
- ***Target:*** El trabajador a quien se le ha solicitado el intercambio.
- ***Offered job:*** El trabajo ofrecido.
- ***Requested job:*** El trabajo solicitado
- ***Accepted:*** Define si ha sido aceptado, rechazado o por decidir en caso de nulo.
- ***Day:*** Día para el que se solicita el intercambio.

Puede parecer innecesario guardar las dos claves candidatas para las tareas (día, trabajador), (día, trabajo), pero es necesario guardar un seguimiento de las solicitudes una vez ya se hayan intercambiado los valores de las tareas.

4.3.1.6 Plan

Especifica los trabajadores disponibles en un intervalo de tiempo y los trabajos pendientes. Podríamos pensar como una analogía la definición de un semestre en esta escuela, marcada por un comienzo y un fin.
Esta entidad es únicamente necesaria para que el planificador sepa qué repartir.

- ***Start date:*** Marca la fecha de inicio del plan.

No se marca la fecha de final del plan ya que se asume va a seguir vigente hasta que comience el plan siguiente. Si verdaderamente se busca finalizar el plan sin que entre en vigencia uno nuevo, quedará la opción de definir un plan vacío sin trabajadores ni trabajos a cumplir.

4.3.1.7 Restriction

La restricción funciona a modo de etiqueta tanto como para trabajadores como para trabajos.

- ***Tag:*** Nombre e identificador de la restricción.
- ***Description:*** Breve descripción opcional de la restricción
- ***Variant* (Forbidding/compulsory)*:*** Un trabajo y un trabajador que compartan una restricción de tipo *Forbidding* serán incompatibles (Por ejemplo, exposición a radiación etiquetaría como *Forbidding* a sala de rayos y empleada embarazada).
  Compulsory sólo permite realizar una tarea a trabajadores que también tengan la etiqueta, como puede ser “alto grado de experiencia” en un quirófano difícil.

4.3.1.8 Preference

Alberga la preferencia de un usuario de realizar determinadas tareas por encima de otras. Con una puntuación de 0-9, siendo 9 la máxima preferencia y 0 la preferencia mínima.

4.3.2 Vistas

Las siguientes vistas no forman en sí parte del modelo, pero son importantes para desglosar funcional y conceptualmente las relaciones del mismo. Deben reflejarse dinámicamente en las vistas las modificaciones que se realicen sobre las entidades y relaciones de las que partan.

4.3.2.1 Incompatibility

Es una capa de abstracción por encima de ***restriction***. Recopila todos los pares trabajador, trabajo que son incompatibles, así como la restricción que los vuelve incompatibles, bien sea por ser una restricción prohibitiva como una restricción de obligatoriedad.

4.3.2.2 ExchangeTasks

Relaciona cada intercambio con todas las tareas que afecte. Esto no implica únicamente las tareas que se hayan solicitado en el intercambio, sino también el resto de tareas afectadas en consecuencia del intercambio, como guardias y descansos.

4.3.2.3 ExchangeExchanges

Esta relación es una capa de abstracción que relaciona los intercambios con los intercambios ajenos a los que afecte por compartir tareas.
Supongamos que para un determinado día, a un determinado trabajador le corresponde realizar una tarea muy popular entre el servicio de anestesia al estar muy bien retribuida.

Siendo este el caso, recibirá numerosas solicitudes de intercambio.

En el momento en el que se acepte el intercambio, cualquier otro intercambio que se encuentre pendiente y tenga como objetivo cualesquiera de las tareas deberá ser rechazado automáticamente.

Esta vista facilita rechazar fácilmente todos estos intercambios dependientes.

4.3.3 Planificador

El planificador es el eje fundamental de la aplicación. De acorde con lo estipulado en el plan, toma los trabajadores disponibles y los reparte entre las tareas teniendo en cuenta las restricciones débiles y garantizando las restricciones fuertes.
Recuérdese que el objetivo es repartir para cada día de un intervalo N trabajadores para M trabajos.

4.3.4 Idea inicial descartada

En una primera aproximación al problema, se consideraban las guardias como una entidad separada de los *trabajos convencionales* al ser el tipo de trabajo más restrictivo y necesario de todos. La guardia localizada no es un trabajo en sí, pero los trabajadores localizados siempre trabajan en reanimación para el hospital que se ha estudiado.

La idea consistía en repartir primero todas las guardias de forma inflexible y rotatoria empleando aritmética modular y una vez asignadas, repartir el resto de trabajos. Del resto de trabajos, una porción se considerarían como "opcionales" para acomodar una posible falta de personal.

Trabajo	**L**	**M**	**X**	**J**	**V**	**S**	**D**
Guardia presencial	Phil	Bob	Ed	Sam	Phil	Bob	Ed
Guardia localizada y REA	Ed	Sam	Phil	Bob	Ed	Sam	Phil

Tabla 3: Rotación de guardias inicial con el mínimo posible de trabajadores (4).

Para el reparto de trabajos convencionales se ideó inicialmente emplear la búsqueda en grafo informada mediante heurística. En concreto el algoritmo A*, donde cada nodo del grafo consistiría en haber asignado un trabajador a un trabajo.
No obstante, un diseño así contenía fallos fundamentales:

- Una gran porción del planificador consistía en ***hard-code***, un importante antipatrón de diseño consistente en incrustar datos directamente o *a fuego*.
- Repartir primero las guardias implicaba partir en dos el planificador, haciendo que la segunda parte tuviese una dependencia fuerte de la primera.
- El número de guardias diarias es inflexible (¡y obligatorio!). En otros hospitales más grandes podrían darse casos de hasta 4 especialistas de guardia en un mismo día.
- Una vez realizado el reparto, la tabla resultante de guardias es inflexible. Si el administrador decidiese hacer cambios a mano, podría fallar el reparto de las guardias.
- Es muy difícil encontrar una heurística viable para describir el estado de un reparto de trabajos, por lo que el rendimiento tenderá al de la fuerza bruta **O(N!)**.

4.3.5 Diseño del algoritmo de planificación actual

El problema del reparto de los trabajos convencionales diarios puede resumirse como el **problema de la asignación** si no tenemos en cuenta el problema del reparto de las guardias. Por ello se ha decidido acomodar el problema tratando las guardias como si fuesen trabajos convencionales como un quirófano o pasar consulta. Esto implica que pueda existir un número arbitrario de guardias a gusto del administrador. En el reparto de costes se explicará cómo se ha solucionado el reparto de las guardias.

Una vez simplificado el problema a N×M, encaja perfectamente con el algoritmo de **Kuhn-Munkres** descrito en el estado del arte. El diseño del funcionamiento del planificador para un día es de este modo:

1. Obtención de los **trabajadores** y **trabajos** para el **plan** al que pertenece este **día**.
2. **Preparación** de una **matriz** de coste con dimensiones N×N representando las posibles combinaciones (trabajador, trabajo). Si el número de trabajadores y trabajos no coincide, se añade una **fila o columna postiza** representando una *"no asignación"*.
3. **Cálculo de los costes** (trabajo, trabajador) a partir de los datos almacenados en el modelo.
4. Aplicación del **algoritmo húngaro**.
5. **Creación de tareas** en la base de datos con las celdas seleccionadas por el algoritmo. No obstante, si el coste del par (trabajo, trabajador) **supera** el **umbral máximo**, significa que de asignarse, se **violaría** una **restricción fuerte**. En tal caso, tanto el trabajo como el trabajador se almacenan en la base de datos como "*no asignado"*.

4.3.6 Cálculo de costes.

El cálculo de costes en este contexto se refiere al cómputo del coste de una única celda en la matriz sobre la que iterará el algoritmo húngaro. Es decir, el coste de asignar un determinado trabajador a un trabajo en concreto para cierto día: el coste de la creación de una tarea.

El reparto de tareas para un día va ligado a **restricciones fuertes**, que no se deben violar de ninguna manera en el reparto y **restricciones débiles,** que son el siguiente objetivo por maximizar. Según está definido el problema, podemos clasificar los siguientes criterios en ambas categorías. Los criterios se ordenan por orden de importancia descendente:

- **Restricciones fuertes:**
    - *Incompatibilidades:* Indica que el trabajador no debe realizar la tarea. Un trabajador es incompatible con todos los trabajos si estuvo el día anterior en una **guardia**.

- ***Restricciones débiles:***
    - *No asignación:* Puede darse el caso de que el número de trabajores y de trabajos no sea el mismo; es el caso de la fila o columna comodín.
    - *Prioridad de trabajos:* Si se dispone únicamente de un trabajador para dos tareas, se descartará la menos prioritaria.
    - *Preferencias y balance:* El reparto de trabajos ha de ser igualitario entre los trabajadores. El balance se obtiene como el ratio de veces que el trabajo en cuestión se ha asignado con anterioridad como tarea entre el número total de trabajos asignados.
      No obstante, los trabajadores pueden tener preferencia de realizar una tarea antes que otra lo cual sesgará la balanza hacia las preferencias de los trabajadores en contra del balance.

Esto significa, por ejemplo, que entre dos asignaciones la preferencia de un trabajador es completamente irrelevante si la prioridad del trabajo es distinta.

Para dar prioridad a un criterio sobre otro, se ha seguido una estrategia de ordenar los valores numéricos de cada criterio dentro del coste como una máscara. Así, al aplicarse el algoritmo húngaro, los bits más significativos se compararán primero y sólamente se tendrán en cuenta los criterios menos importantes cuando los más importantes tengan el mismo valor.

Por ejemplo, el coste de asignar un trabajo a un trabajador que es incompatible, cuya prioridad es 3, que lleva haciendo el 50% de su trabajo y tiene una preferencia mínima de 0 sería **23.75**. El desglose de la máscara es el siguiente:

Incompatible: Sí. (binario)	Sin asignación: No. (binario)	Prioridad del trabajo: 3. (Base 10)	Preferencias y balance: 0.75
Decenas		Unidades	Decimales
1	0	3	.75

Tabla 4: Representación en máscara para el coste.

En cuanto al equilibrio entre balance y preferencias, se computa restándole al ratio de balance el valor de la preferencia ajustado con el peso especificado en las opciones de la aplicación. (Sin olvidar la normalización del valor al rango [0-1]).

$$cost(w,j,d) = 20 \times esIncompatible(w,j,d) + 10 \times esComodin(w,j,d) + prioridad(w) + \frac{\frac{asignado(w,j)}{totalAsignado(w)} - \frac{preferencia(w,j)}{10} \times PesoPreferencia + 1}{2}$$

Figura 7: Fórmula para el cálculo de costes

4.3.7 Rutina de planificación

Suponiendo que la aplicación acaba de desplegarse, la lógica de funcionamiento es la siguiente:

1. El administrador hace *log-in* y entra en la **sección de administración**.

2. Da de alta a la plantilla del hospital como **trabajadores**.

3. Registra los distintos **trabajos** (p ej. quirófanos).

4. Especifica el **plan** o planes y el comienzo de cada uno. En el plan se selecciona los **trabajadores** y **trabajos** involucrados en dicho **periodo de tiempo**.
   El administrador ha de tener en cuenta que la plataforma va a generar por defecto 100 días desde el día actual.

Los pasos anteriores solamente deberían ser necesarios una vez. En cualquier momento el administrador puede añadir o alterar planes o trabajadores y trabajos sin la necesidad de detener la aplicación.

5. Se invoca la rutina automática del planificador a una hora determinada cada día (o en otro intervalo de tiempo, o manualmente).

6. Desde el día actual genera tantos días vengan especificados a pregenerar desde las opciones de la aplicación. Si ya existen tareas en este intervalo, genera únicamente las que falten a partir de la última.
   Por defecto, el planificador se configura para que actúe diariamente, por lo que cada día **d** se generarán las tareas del día **d+100**.

En cualquier ocasión podría el administrador modificar a mano las asignaciones que ha realizado el algoritmo, incluso si implica saltarse las normas. El algoritmo va a seguir funcionando sin problemas.

4.4 Realización de intercambios

Una vez repartidas las tareas, siempre cabe la posibilidad de que los trabajadores se intercambien las tareas. Los intercambios se realizarán siempre entre dos trabajadores en un mismo día.

Supongamos el siguiente caso práctico: disponemos de los trabajadores Alicia, Bob y Carmen, con las tareas asignadas de guardia, quirófano de obstetricia y quirófano digestivo respectivamente.

Partamos también del conocimiento de que las guardias están muy bien pagadas y de que a Alicia le encanta la obstetricia.

El proceso desde que solicita el intercambio hasta que se realiza es el siguiente:

1. Tanto Bob como Carmen entran en la aplicación desde un ordenador o dispositivo móvil y ven la planificación de la semana. Ahí se muestra que Alicia tiene guardia el viernes, especialmente bien pagado por el día de la semana.

2. Ambas personas **pinchan** sobre la tarea para **solicitar** un **intercambio**.

3. El controlador toma inmediatamente cada petición y la envía al modelo.

4. El **modelo verifica** que la **solicitud** es **correcta** y que se puede realizar sin violar restricciones fuertes. De no ser así, se notifica al trabajador que no se puede solicitar el intercambio.

5. El **modelo registra** la **solicitud** de intercambio dentro de la base de datos, marcándola como *pendiente.* Las solicitudes realizadas las podrán ver desde la **bandeja de salida de intercambios**.

6. **Alicia** entra en la aplicación y ve que en su **bandeja de entrada de intercambios** hay dos nuevos intercambios pendientes, ambos para su mismo trabajo.

7. Si **Alicia rechazase** un intercambio, lo único que el modelo debería de hacer es marcarlo como rechazado. No obstante, en esta situación **Alicia acepta** la solicitud de **intercambio** de **Carmen** (quirófano de obstetricia).

8. Una vez más, el **modelo verifica** que la **solicitud** es **correcta**.

9. Los **intercambios** que se vean **implicados** por este intercambio se **rechazan** automáticamente. Este incluye el de **Bob** pero también los relacionados pendientes de **Carmen.**

10. De las **tareas** implicadas, se **intercambian** los trabajadores: **Alice** y **Carmen**. Al tratarse de una **guardia**, **también** se **intercambia** el **descanso** del día siguiente de Alice con la tarea de Carmen. Las tareas que pudiesen haberse visto implicadas en el intercambio de este día posterior, también se han rechazado en el paso 9.

11. Los cambios se ven reflejados en las bandejas de intercambio de Alice, Bob y Carmen, así como en el calendario.

Verificaciones que se realizan al formular o aceptar intercambios

- La tarea no ha ocurrido ya.
- No se puede solicitar el mismo intercambio si ya existe (salvo rechazado).
- No se puede aceptar un intercambio si no está pendiente.
- El intercambio no se está haciendo consigo mismo.
- Las tareas solicitadas existen y coinciden con el intercambio.
- El intercambio no implica múltiples guardias en cascada.
- En caso de guardia, el trabajador figura el día siguiente en el plan (descansando). Esto es porque podría darse el caso de cambio de plan justo entonces con trabajadores distintos y es obligatorio descansar.
- Las nuevas tareas pendientes por asignar no relacionan trabajos con trabajadores que aparezcan en la tabla de incompatibilidades.

4.5 Vistas

Vistas son la parte de la aplicación que toma los datos una vez formateados y los muestra por pantalla. Son conocidas como *templates* en django ya que de forma similar a como funciona PHP, el fichero es una plantilla HTML en la que se incrusta código que será rellenado por el controlador.

Ya que la aplicación ha de ser accesible desde cualquier dispositivo (ordenador o móvil), ha sido necesario desarrollar una interfaz gráfica que se amoldase a la resolución del aparato para un uso cómodo.

A continuación se describirá la apariencia general de la plataforma. Es ***importante*** ver el anexo para ver todas las secciones que componen la página. A la izquierda, cómo se vería en un móvil. A la derecha, cómo se vería en un monitor de resolución relativamente baja (1024x768).

La estructura básica de las páginas es la siguiente:

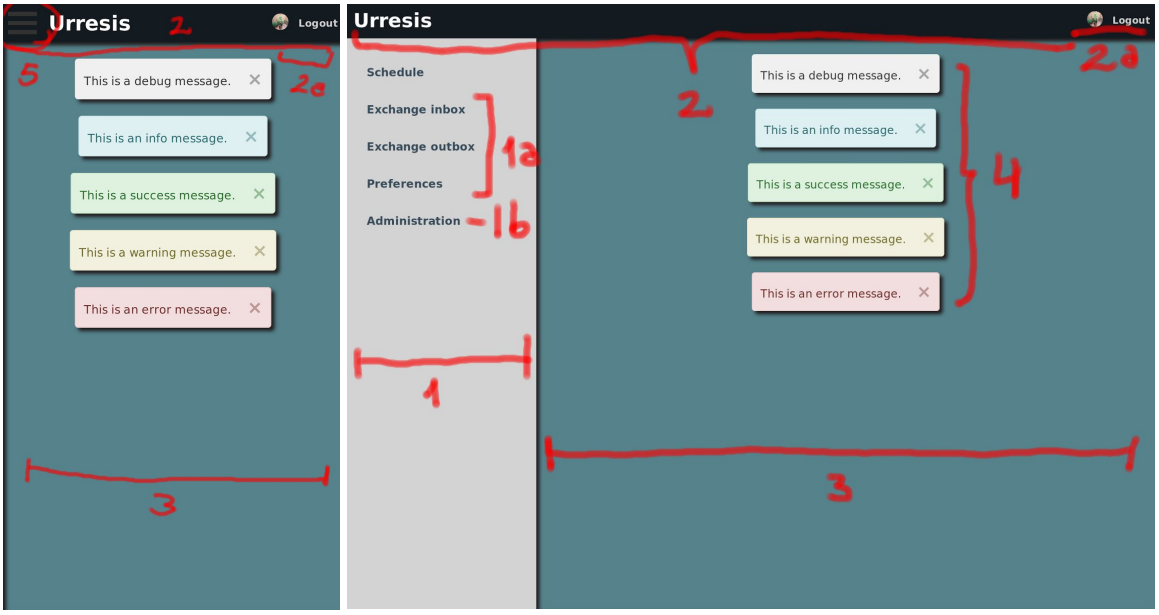


Figura 8: Estructura básica de la página.

1. Barra lateral de navegación. Si la pantalla es demasiado pequeña, se colapsa para dejar espacio al contenido.
    a. Intercambios y preferencias. Estas entradas del menú desaparecen si el usuario no tiene asociado un trabajador.
    b. Al portal de administración. Visible sólo por administradores.
2. Cabecera de la página. Contiene el logotipo.
    a. Enlace al perfil
3. Contenido de la página. Esto es lo que cambia en cada sección.
4. Notificaciones. Aparecerán siempre por encima del contenido. Se pueden cerrar pinchando sobre la cruz y emplean distintos colores dependiendo del tipo.
5. Conmutador para mostrar/ocultar la navegación si no cabe.

5 Desarrollo

En esta sección se describirá de forma detallada cómo se han implementado las distintas funcionalidades especificadas en la sección anterior de diseño desde una perspectiva más cercana al código, los problemas que han surgido y cómo han sido abordados.

5.1 Tecnologías empleadas

- **Aplicación web:** Es la solución más extendida y multi-plataforma en este momento. Asegura su funcionamiento en cualquier entorno y con la flexibilidad de poder utilizar un amplio abanico de frameworks y lenguajes.

- **Python:** Es un lenguaje de programación muy maduro para cualquier propósito por lo que tiene un sinfín de librerías disponibles. A pesar de ser un poco lento, existen implementaciones de python para hacerlo más rápido y se le puede anclar código en C/C++ sin excesivas dificultades.

- **Django:** Se ha escogido al seguir el esquema modelo/vista/controlador, emplear una sintaxis limpia, haber sido desarrollado durante más de 10 años, poseer una gran comunidad y tener una documentación extensa y detallada.

- **SQLite:** Este RDBMS está disponible desde python sin necesidad de instalar ni configurar nada. Desde bash basta con ejecutar el comando sqlite3 seguido del fichero almacenando la base de datos y ya se pueden mandar queries. Se ha empleado para el desarrollo de gran parte de la aplicación hasta haber dado con sus limitaciones.

- **PostgreSQL:** El gestor de bases de datos 'serio' sobre el que se despliega el producto final. Ha sido escogido por adherirse al máximo al estándar SQL y por su licencia libre.

- **HTML:** Al ser SGML un lenguaje incómodo de manejar, se ha valorado generar las páginas empleando YAML. No obstante, esto supondría tener que "compilar" cada vez que se realizase un cambio en vez de simplemente programar sobre las plantillas de django.

- **Flexbox:** Con el nuevo estándar de CSS se pueden estilizar las páginas para hacerlas responsivas para su usabilidad en el móvil. Así no es necesario crear una aplicación por separado para cada plataforma móvil y no se paga el precio de abotargar la página con múltiples capas de abstracción en base al uso de distintos **frameworks front-end** (especialmente aquellos que empleando javascript de forma extensiva.
  Se consideró el estándar aún más nuevo de **grid** al ser más completo y manejable pero su grado de adopción no es aún lo suficientemente alto. Flexbox tiene una adopción de prácticamente el 100% en los navegadores utilizados hoy en día.

- **CSS básico:** Se ha valorado emplear lenguajes de preprocesado como **SASS** o **LESS** al ser más limpio y organizado con variables, pero se han descartado como opción al suponer el mismo problema que usar YAML: el recompilar cada vez.

5.2 Entorno de trabajo

Para el desarrollo del proyecto se ha utilizado un portátil personal con la distribución linux Fedora Rawhide (versión 27) por poder utilizar las herramientas más nuevas (notoriamente, el poder utilizar python3.6 en vez de 3.5 y vim versión 8 en vez de la versión 4).
Los paquetes de python han sido instalados con la herramienta pip3 a nivel usuario en la carpeta $HOME. Para el producto final, se empaquetará en un entorno virtual o *virtualenv*.

Para la edición de código se ha utilizado el editor de texto **Vim**.
No tiene tantas prestaciones como la de un IDE completo como pueda ser PyCharm pero ahorra la necesidad de tener que configurar el proyecto.
No obstante, es altamente flexible y posee funcionalidad de formateado y completado de código, coloreado de sintaxis o poder delegar funcionalidad a otras aplicaciones.

Para guardar copias de seguridad se ha empleado versionado con **git**. Esto ha sido especialmente útil y ha permitido dejar cierto trabajo de lado para continuar trabajando en una funcionalidad distinta en otra rama y descartar trabajo inútil fácilmente revirtiendo a un estado anterior, así como para encontrar cuándo han aparecido determinados bugs.
Como repositorio online se ha empleado **bitbucket** al ser gratuito y privado.

5.3 Jerarquía de ficheros

media → Almacenamiento de recursos dinámicos. Avatares subidos por el usuario.
tfg → Carpeta del proyecto
├── settings.py → Configuración (conector a BBDD, isolation level, etc.).
├── static → Recursos estáticos del proyecto.
├── templates → Plantillas (vistas) externas a la aplicación (login, admin...).
├── urls.py → URL dispatcher. Importa las de urresis y de administración.
└── wsgi.py → Módulo para poder desplegar el servidor.
urresis → Carpeta de la aplicación
├── admin.py → Configuración y personalización de la página de administración.
├── apps.py → Constantes de la aplicación (p. ej: preferencia de tarea por defecto)
├── forms.py → Formularios a renderizar en el HTML.
├── management → Comandos CLI (expone el planificador al sistema operativo).
├── migrations → Deltas de cambios de las tablas de la base de datos.
├── models.py → Alberga el modelo (incluyendo la lógica, no sólo el ORM)
├── planifier.py → Planificador Kuhn-Munkres.
├── static → Recursos estáticos de la aplicación (js, css, imágenes…)
├── templates → Plantillas (vistas) de la aplicación.
├── templatetags → Extensiones de etiquetas para las vistas.
├── tests.py → Batería de pruebas.
├── urls.py → URL dispatcher. Aquí se definen las rutas y mapeo a las vistas.
└── views.py → Controladores.

5.4 Modelo y ORM

La finalidad del Object-Relational Mapping es conseguir representar las distintas entradas en la base de datos como objetos python. Esto permite tratar con mucha facilidad la información, ya que python es un lenguaje de programación muy potente.
De hecho, una de las ventajas de este modelo es poder integrar lógica de negocio dentro de los propios objetos, minimizando el código que llevaría el controlador. De este modo se produce código de acoplamiento bajo y alta cohesión, perfectamente modular.

5.4.1 Creación del modelo

Las distintas entidades del modelo se definen en *models.py* extendiendo la clase *Model* de django. Veamos por ejemplo la definición de la entidad Tarea en el modelo:

```
class Task(models.Model):
    job = models.ForeignKey(Job, null=True, blank=True,
                            on_delete=models.SET_NULL)
    date = models.DateField()
    worker = models.ForeignKey(Worker, null=True, blank=True,
                            on_delete=models.SET_NULL)

    class Meta:
        unique_together = (('job','date')), ('worker','date'))
        get_latest_by = 'date'

    def __str__(self):
        return '{}: ({}|{})'.format(self.date, self.worker, self.job)
```

Figura 9: Implementación de Tarea en models.py

Cada atributo de la clase (job, date, worker) se conoce como *Field* y tiene una correspondencia 1-1 con las columnas de la base de datos. Entre los tipos de datos existe una variedad superior que en los gestores RDBMS que django materializa en la base de datos dependiendo de a cual se dirija.

Por ejemplo: SQLite no soporta el tipo de dato booleano, por lo que django define la columna en la base de datos como un entero.

Existen también otros campos exclusivos de django, como el campo *ImageField* utilizado por la entidad *Worker*, que guarda la ruta de la imagen del avatar.

Las relaciones entre entidades no se definen como entidad nueva, sino que se dispone de los siguientes *Fields:* **ForeignKey, ManyToMany, OneToOne.** Django se encargará de crear las tablas intermedias, lo cual es muy útil para aquellos que desconozcan el modelo ER.

Cada entidad posee internamente la clase *Meta,* donde se define información relevante al modelo, como los *constraints* de la base de datos.

En este ejemplo en concreto de Tarea, uno puede preguntarse por qué no está definida la clave primaria. Esto se debe a una de las graves limitaciones del ORM de django: no soporta claves compuestas.

De hecho, django utiliza claves subrogadas para absolutamente todo, incluyendo las tablas de unión; que únicamente necesitan la clave de las dos entidades que relacionan.

```
select * from urresis_worker_restrictions;

 id | worker_id |    restriction_id
----+-----------+----------------------
  2 |         6 | generic restriction X
  3 |         6 | Generic obligation Y
  4 |         3 | generic restriction X
  5 |         1 | Generic obligation Y
(4 rows)
```

Figura 10: Tabla de unión entre trabajador y restricciones con clave subrogada innecesaria

Además de los modelos, django guarda en la base de datos una captura del estado de *models.py*. Realizar cambios en el modelo es como sigue:

1. Se añade un campo adicional en el modelo.
2. Se ejecuta ./**manage.py makemigrations**. Con esto django compara el fichero models.py con lo que hay en la base de datos y genera en el módulo *migrations* el código SQL para añadir dicha columna.
3. Se aplican los cambios a la base de datos mediante **./manage.py migrate.**
4. Se resuelven interactivamente los problemas que puedan surgir, como rellenar los campos de las nuevas columnas si no pueden ser nulos y no se ha especificado un valor por defecto.

5.4.2 Interacción con el modelo

Para poder emitir consultas, se ha de importar los modelos que vayamos a utilizar mediante código python. Las consultas emplean el lenguaje definido por el ORM de django.
Para obtener todas las tareas asignadas a Juan hoy, la sintaxis es la siguiente:

```
tareas_juan_hoy=Task.objects.filter(worker__user__name='Juan',
day=datetime.today())
```

Como se puede observar, django hace automáticamente el join de tablas (task-worker-user) para filtrar los resultados. El objeto resultante en tareas_juan_hoy es denominado *queryset*, una evaluación perezosa de los resultados, donde cada entrada se corresponde con una fila. Evaluación perezosa significa que la consulta a la base de datos no se realiza hasta que no se itera sobre el *queryset*.

ModelManager

ModelManager está almacenado en el atributo *objects* contenido de cada clase del modelo. Es lo que interactúa con la base de datos; la lógica de dominio. Contiene todos los métodos que actúan a nivel de tabla (select, update, delete…).

Sabemos que la página del plan semanal lista los nombres de todos los trabajadores. Esta información no se encuentra en la tabla del trabajador sino en el usuario que tiene asociado.
Dentro de la clase del modelo *Worker* podríamos definir un atributo accesor que retornase ***self.user.name***. Esto realmente no es una buena idea, ya que requiere acceder a la base de datos para obtener el nombre del usuario: en el caso de la página del planning semanal, se mandaría una consulta para obtener todos los trabajadores y luego una consulta adicional por cada uno de los nombres que se va a listar.

Para evitar esta situación, se emplea un *ModelManager* extendido denominado **WorkerWithNameManager** que toma el *queryset* original y le añade la columna de nombre que posteriormente se mostrará. El nombre del trabajador toma el primero de estas opciones:

Nombre completo → Nombre incompleto → Nombre de usuario → ID de trabajador

Dentro del *ModelManager* no sólamente se extienden los *querysets*. También es el sitio idóneo para implementar la lógica de dominio.

ExchangeManager es el *ModelManager* extendido del modelo ***Exchange***. En él se implementa la lógica para solicitar, aceptar y rechazar intercambios tal y como se ha definido el funcionamiento en el apartado de diseño. De tal modo, el modelo Exchange expone los siguientes métodos:

- Exchange.objects.accept(exchange)
- Exchange.objects.decline(exchange)
- Exchange.objects.request(by, to, day, offer, request)

Así, cuando el usuario acepta, rechaza o formula un intercambio, el controlador únicamente tiene que invocar cualquiera de estos métodos por lo que se aligera la cantidad de código del mismo.

En caso de que la petición no sea correcta, los métodos arrojan una excepción de validación, que será capturada en el controlador para añadir en la vista manejadora un mensaje de error que se mostrará al usuario.

5.4.3 Limitaciones del ORM

El ORM de django es muy potente. Permite operaciones de **select**, **update**, **delete**, **filtrado**, **agregación**, **álgebra de conjuntos**, **ordenado, casewhen**…
La sintaxis puede volverse algo aparatosa al confeccionar consultas más complejas, pero se puede utilizar el objeto Q para encapsular porciones de la consulta y aplicar operaciones a nivel de bits. Recuérdese que los querysets se pueden encadenar al ser evaluados perezosamente:

```
#Check if either worker is on duty today but not listed tomorrow!
q_either_worker = Q(id=requested_by.id) | Q(id=requested_to.id)
workers_tomorrow = Worker.objects.filter(task__date=nextday)
duty_today_missing_tomorrow = Worker.objects.filter(q_either_worker,
                                ~Q(id__in=workers_tomorrow),
                                task__date=day,
                                task__job__onduty=True)
```

Figura 11: Consulta compleja empleando el objeto Q

Pero a pesar de todo, cuando la consulta se vuelve suficientemente compleja, hay cosas que el ORM simplemente no puede hacer. Esto es una limitación que reside en el algoritmo de *joining* de tablas basado en el uso de claves subrogadas para todo.

El problema se descubrió al intentar escribir el código para seleccionar los intercambios que se ven afectados por otros intercambios, destapando funcionalidad aparentemente imposible con el ORM de django: cruzar tablas por múltiples columnas, self-join y cruzar con una misma entidad múltiples ocasiones en una misma consulta con condiciones distintas.

Para intentar solventar este problema, django posee el método **raw** integrado en Model.objects.**raw**(‘select * from table;), que permite construir consultas “crudas”.

Desafortunadamente, **raw** no es una solución adecuada ya que su integración con el resto de django no es nada buena. No se puede conectar con la mayoría de componentes de django (como para la construcción de formularios), no se puede mezclar con el ORM, las excepciones ocurren en situaciones descontroladas…

5.4.4 Solución mediante vistas SQL

Dentro de la documentación interna de django se especifica la opción de bajo nivel **managed=False** que se le puede atribuir a la clase **Meta** de cada clase de modelo, lo cual le indica a django que no debe de hacer nada en la base de datos y no lo tenga en cuenta para generar migraciones.

La solución consiste en crear manualmente vistas SQL inyectando el código dentro de los ficheros de migraciones de django. Al mismo tiempo se crea una clase no-manejada en

models.py, donde las relaciones se marcan como *on_delete=DO_NOTHING* para que django no intente borrar entradas en la vista. Asimismo es necesario (por el funcionamiento interno de django) crear una columna ID en cada modelo. Django la emplea para la modificación de la tabla; como la vista no se va a intentar alterar, esta columna puede tomar cualquier valor arbitrario (como 0).

```
class Incompatibility(models.Model):
    """This is is the view lists incompatible jobs according to restrictions """
    worker = models.ForeignKey('Worker', on_delete=models.DO_NOTHING)
    job = models.ForeignKey('Job', on_delete=models.DO_NOTHING)
    restriction = models.ForeignKey('Restriction', on_delete=models.DO_NOTHING)

    class Meta:
        managed=False
```

Figura 12: Modelo 'falso' funcionando sobre una vista SQL

Esta clase de sólo lectura se puede manejar de la misma manera que el resto y libera de tener que emplear el ORM: *Incompatibility.objects.filter(worker=worker, job=job).exists()*

```
drop view if exists urresis_incompatibility;
create view urresis_incompatibility as
with unfulfilled_requisites(worker, restriction) as (
        select w.id as worker, r.name as restriction from urresis_worker w
        cross join urresis_restriction r
        left join urresis_worker_restrictions wr
        on w.id = worker_id and r.name = restriction_id
        where r.variant = 'C'
        and worker_id is null),

    forbidden(worker, restriction) as (
        select worker_id, restriction_id from urresis_worker_restrictions wr
        inner join urresis_restriction r
        on wr.restriction_id = r.name
        where variant = 'F'),

    restricted(worker, restriction) as (
        select worker, restriction from unfulfilled_requisites
        union
        select worker, restriction from forbidden),

    incompatible(worker_id, job_id, restriction_id) as (
        select worker, job_id, restriction from urresis_job_restrictions
        inner join restricted
        on restriction = restriction_id)
--ID = 0 because django uses an ID column, but we don't actually need it
select 0 as id, worker_id, job_id, restriction_id from incompatible
```

Figura 13: Vista SQL integrada en las migraciones para acoplamiento en ORM

5.5 Controlador

El controlador es la capa intermedia que procesa las peticiones del usuario, interactúa con el modelo y prepara las vistas para mostrar la página. Es el motor en sí.

Cuando un usuario solicita una dirección, el motor busca en el fichero *urls.py* las diferentes rutas en base a expresiones regulares. Cada ruta tiene asociada un controlador al que se le mandará el *request* solicitado por el navegador del usuario para que lo procese.

5.5.1 Controladores predefinidos por django

Para la autenticación de usuarios existe en django el paquete django.contrib.auth, que contiene controladores predefinidos que manejan de forma segura el proceso de registro y login, así como un modelo User donde almacena en la base de datos usuario, contraseña y otros datos relevantes al inicio de sesión.
Adicionalmente, django posee la conocida como *admin site*, una sección desde la que el administrador puede realizar tareas de tipo CRUD (Create, Read, Update, Delete) sobre los modelos de la aplicación.
Para ello se han registrado en el fichero *admin.py* las entidades Trabajador, Trabajo, Tarea, Plan y Restrición, que son sobre los cuales el administrador debe de tener acceso manual de cambios.

5.5.2 views.py

El eje fundamental del funcionamiento del controlador se encuentra en las distintas funciones definidas en *views.py*

Dentro de *views.py* hay definidas una serie de funciones (y clases) que reciben como parámetro un objeto HTTPRequest conteniendo aparte de cabeceras, cookies y parámetros GET/POST, también información introducida por el middleware, como la instancia User conectada.

Todos los controladores definidos en *views.py* están protegidos por *login*. En el caso en que un usuario trate de acceder a una ruta protegida, será redirigido primero a la página inicial de *login* de la aplicación */accounts/login/*

Las funciones definidas son las siguientes:

- **Clase WeekTaskView:** Extiende la clase genérica *WeekArchiveView* de django, sobreescribiendo una serie de métodos para que averigüe la fecha actual si no se han especificado los parámetros de semana y año.
  Prepara el rango de las tareas de una semana, que envía a la vista principal del calendario semanal.

- **PasswordChange:** En caso de GET, genera el formulario para el cambio de contraseña. En caso de recibir los datos de cambio de contraseña por POST, verifica que las contraseñas coincidan y acutaliza la contraseña del usuario conectado.

- **Profile:** Prepara la vista para mostrar el perfil. Desde aquí cotrola si el usuario tiene trabajador asociado o no para preparar el contenido del trabajador o no. En caso de POST verifica los campos y acutaliza.

- **Exchanges:** Controla tanto la vista del buzón de entrada de intercambios como del buzón de salida. Prepara ligeramente el modelo Exchange antes de pasarle la vista.

- **Exchange:** Controla la lógica cuando el usuario pincha en aceptar o denegar en el listado de intercambios. Captura las distintas excepciones de validación que pueda generar el modelo para mostrarlos en la siguiente vista.

- **Exchange_request:** Recibe como parámetro la tarea que se pretende intercambiar que ha seleccionado el trabajador. A partir de ahí confecciona el paquete que contiene el día, los trabajadores implicados y los trabajos implicados y prepara una vista para mostrarlo por pantalla. Cuando el usuario pinche submit, toma este paquete y solicita en el modelo el intercambio.

- **Preference:** Utiliza el patrón *factory* para generar tantos *forms* como preferencias tenga el usuario. Cuando el usuario pulse submit, guarda los cambios tras comprobar casos anómalos como que el usuario haya puesto dos veces el mismo trabajo.
  En la vista en la práctica utiliza un único *form* con identificadores para cada entrada.

5.5.3 Rutina del planificador

Dentro del paquete management está definido el comando CLI **taskgen**. Este comando recibe opcionalmente como parámetro el número de días que han de ser pregenerados a partir de hoy. En el caso de que el parámetro no se especifique, obtendrá este valor de la constante predefinida en el fichero de configuración de la aplicación *apps.py*

A partir de esa información, la rutina invoca al planificador para que genere cada uno de los días necesarios. Para el reparto, el planificador emplea las herramientas *numpy* y *scipy* para aplicar el **algoritmo húngaro** con un rendimiento óptimo.

Esta rutina está diseñada para que sea invocada periódicamente (cada día o cada semana) por las tareas periódicas del sistema operativo, como puedan ser CRON, systemd, o Task Scheduler en máquinas Windows.

5.6 Vistas

Las vistas son la parte de la aplicación encargadas de generar el *HTML* que se va a mostrar al usuario, junto a los recursos de *Javascript* y *CSS*. Las vistas sólo contienen el mínimo posible de lógica para poder mostrar el contenido que ya ha sido procesado por el controlador.

Similarmente a como ha funcionado siempre *PHP,* las vistas en django son ficheros HTML con fragmentos de código con una sintaxis especial que será evaluado a la hora de servirse; de ahí el nombre que les da django de *template*, son plantillas HTML.

```
{% extends "urresis/base.html" %}

{% block extra_style %}
{% load static %}
<link rel="stylesheet" type="text/css" href="{% static 'urresis/request_exchange.css' %}" />
{% endblock %}

{% block main %}
<div class="window">
<div class="window_header">Exchange request</div>
<div class=date>{{ hiddenform.date.value }}</div>
<div class="window_content">
<form method="post">
        {% csrf_token %}
        {{ hiddenform }}
        <div class=exchanges>
                <div class=offering>
                        <img class="avatar" src="{{requester.avatar_url}}" />
                        <div class=worker>{{ requester }}</div>
                        <div class=job>{{ hiddenform.requester_job.value }}</div>
                </div>
                <div class=requesting>
                        <img class="avatar" src="{{target.avatar_url}}" />
                        <div class=worker>{{ target }}</div>
                        <div class=job>{{ hiddenform.target_job.value }}</div>
                </div>
        </div>
        <div class='buttons'>
                <input class='button primary' type="submit" value="Request exchange" />
                <a class="button" href="{% url 'urresis:base' %}">Cancel</a>
        </div>
</form>
</div>
</div>
{% endblock %}
```

Figura 14: Request_exchange.html

La mayoría de vistas tienen una estructura similar a la anterior. Extienden de la plantilla base, de la que heredan el layout principal con su estilo, al que añaden el suyo propio.

El contenido principal suele disponerse en ventanas, con cabecera y cuerpo.

Los formularios están protegidos con un token csrf que lo protege contra *cross site request forgery*. La lógica del formulario en la cual se incluye su saneamiento no se encuentra aquí, sino en el fichero *forms.py* formando parte del controlador.

Todo el aspecto de la plataforma ha sido diseñado a mano empleando flexbox, del nuevo estándar CSS3 para crear una página completamente responsiva. Las vistas de administración ya vienen definidas por django y se ha empleado un plugin responsivo.

6 Integración y pruebas

Uno de los puntos fuertes de django es la batería de pruebas, que está basado en el framework de pruebas de la librería estándar de Python unittest.

Las pruebas se encuentran en el fichero *tests.py*, estructuradas en conjuntos de pruebas similares como métodos de clases que heredan de *unittest.TestCase*. Dicha clase posee el método *setUp,* que prepara la prueba para el resto de métodos.

Al ejecutar el fichero mediante *./manage.py test,* django ejecuta las diferentes pruebas ahí definidas. Sabiendo que el número de pruebas puede ser elevado y de distintos tipos, también se puede filtrar y seleccionar por categorías las pruebas que van a ejecutarse.

Aparte de la funcionalidad ofrecida por *unittest,* django ofrece la clase *Client* para simular la conexión de usuario y ver qué respuestas obtiene ante determinadas peticiones. No es tan completo como *Selenium*, pero sirve para ver el contexto de distintas vistas y a qué rutas tiene acceso un determinado usuario.

Dada la complejidad del código y los numerosos puntos posibles de fallo que puedan ocurrir con dificultades enormes para trazar el error, en vez de diseñar una batería de pruebas para probar el sistema una vez implementado, se ha optado por una estrategia distinta.

Cada fragmento de código que se ha ido implementando de manera incremental, a su misma vez ha sido probado. Pero no ha sido probado únicamente ejecutando código y viendo la correción de los resultados, sino que además se ha integrado la ejecución y el resultado esperado como una prueba en *tests.py*.

Por ejemplo, el planificador inicialmente sólo repartía N trabajadores entre N tareas. Se escribió una prueba para ver que funcionase. Posteriormente repartía N entre M. Se escribió otra prueba. Igualmente con el añadido de preferencias, de guardias, cálculo de pesos…

Esta forma de desarrollo ha sido especialmente útil a la hora de construir queries complejas que se pudiesen separar en partes. Así para cada query existen múltiples pruebas incrementales que la prueban parcialmente, así como una final que la prueba al completo.

A pesar de dotar de una cobertura completa disponible en todo momento desde *tests.py*, esta forma de implementación de pruebas acarrea el problema de que conjuntos de pruebas dejan de ser válidos si hay cambios de funcionalidad. De todos modos, se pueden borrar fácilmente las que fallen y reciclar las útiles.

Aparte de las pruebas de código, también se ha verificado la interfaz y su comportamiento al redimensionar el tamaño del navegador, fundamentalmente sobre Firefox dada la versatilidad de uso del *Responsive Design Mode*. También se ha probado sobre Chrome y sobre un móvil Android.

7 Conclusiones y trabajo futuro

7.1 Conclusiones

En este Trabajo de Fin de Grado se ha desarrollado una aplicación web basada en django y flexbox con el objetivo dotar de una plataforma automatizada mediante el algoritmo húngaro al servicio de anestesia hospitalario desde la cual poder gestionar la rotación de turnos. En el desarrollo de la aplicación se ha llegado a las siguientes observaciones:

- Django es un *framework* demasiado grande que trata de cubrir excesiva funcionalidad. Esto es muy útil si se quieren desarrollar aplicaciones simples basadas en *CRUD*, ya que en tal caso el esfuerzo invertido en desplegar una aplicación así es mínimo. Por el contrario, cuando la complejidad de la aplicación crece, el tamaño del *framework* se vuelve un problema al tener que lidiar con una infinidad de capas de abstracción para amoldar django al problema.

- La tecnología *ORM* se vende como la panacea de unión entre programación orientada a objetos y *SQL*. A pesar de probar ser sencillo e ideal para consultas simples, ha demostrado ser inviable cuando las consultas son muy complejas.
  Su uso está especialmente no justificado en python, donde los objetos son completamente dinámicos y en ningún momento hay que definir sus atributos.

- El algoritmo **Kuhn-Munkres** ha sido eficaz para solucionar el reparto pese a las restricciones y su eficiencia de rendimiento ha superado con creces las expectativas. Aunque las soluciones basadas en machine-learning sean más adecuados para solucionar problemas mucho más complejos, no hay que descartar los modelos matemáticos clásicos de optimización exacta.

- Flexbox ha sido una herramienta fantástica para organizar el *layout* de la página. No es necesario recurrir a *frameworks front-end* ni *hacks* para obtener una página con un aspecto comparable a los estándares de las aplicaciones comerciales. Los nuevos estándares de W3C Flexbox, Grid, Service Workers y web-assembly auguran un futuro brillante a la web, lejos de los apaños empleados hasta ahora.

7.2 Trabajo futuro

Para la plataforma existen los siguientes de expansión:

- Migrar de django a otro *framework* más simple como flask e instalar ahí los complementos necesarios.
- Probar a integrar sistemas de *machine learning* en el reparto y hacerlo más avanzado, como distribuyendo los trabajadores en franjas horarias que puedan solapar.
- Reescribir todo lo implementado en ORM a SQL plano para facilitar su entendimiento.
- Probar la siguiente iteración de tecnologías propuestas por W3C, especialmente Grid para el layout y Service Workers para integrar con notificaciones.

Referencias

[1] J. Canet et al. Antecedentes, objetivos y método de la encuesta de actividad anestésica en Cataluña (ANESCAT 2003)

[2] Dr. José Olarra. Memoria del Servicio de Anestesia y Reanimación. Hospital universitario de Fuenlabrada (2012)

[3] J. Canet et al. Modelo de cálculo de plantillas de los servicios de anestesiología, reanimación y terapéutica del dolor. (Rev. Esp. Anestesiol. Reanim. 2001; 48: 279-284)

[4] F. Bent et al. Personaleinsatzplanung in der operativen Anästhesie. Strukturierte Interviews mit 23 personalverantwortlichen Oberärzten. Anaesthesist 2016 · 65:337–345 DOI 10.1007/s00101-016-0164-5 Springer-Verlag Berlin Heidelberg 2016

[5] H.W. Kuhn, The Hungarian Method for the Assignment Problem, Naval Research Logistics Quarterly 2 (1955) 83–97. ftp://nozdr.ru/biblio/kolxo3/M/MOac/Junger%20M.,%20et%20al.%20(eds.)%2050%20years%20of%20integer%20programming%201958-2008..%20From%20the%20early%20years%20to%20the%20state-of-the-art%20(Springer,%202010)(ISBN%203540682740)(811s)_MOc_.pdf#page=46

[6] James Munkres, Algorithms for the assignment and Transportation Problems, Journal of the Society for Industrial and Applied Mathematics, Vol.5, No.1 (Mar., 1957), 32-38 https://www.math.ucdavis.edu/~saito/data/emd/munkres.pdf

[7] http://www.math.harvard.edu/archive/20_spring_05/handouts/assignment_overheads.pdf

[8] CSS Flexible Box Layout Module Level 1 https://www.w3.org/TR/css-flexbox-1/

[9] CSS Grid Layout Module Level 1 https://www.w3.org/TR/css-grid-1/

[10] Media Queries https://www.w3.org/TR/css3-mediaqueries/

[11] https://tutorialzine.com/2016/12/the-languages-frameworks-tools-you-should-learn-in-2017

[12] http://troels.arvin.dk/db/rdbms/

[13] http://www.itbusinessedge.com/slideshows/top-five-nosql-databases-and-when-to-use-them-07.html

[14] https://www.simple-talk.com/sql/t-sql-programming/questions-about-t-sql-transaction-isolation-levels-you-were-too-shy-to-ask/

Glosario

ACID	Atomicity Consistency Isolation Durability
API	Application Programming Interface
BBDD	Bases de Datos
CLI	Command Line Interface
CRUD	Create Read Update Delete
CSS	Cascading Style Sheets
HDFS	Hadoop Distributed FileSystem
HTML	HyperText Markup Language
IBM	International Business Machines
JSON	JavaScript Object Notation
LAMP	Linux Apache MySQL PHP
LESS	LE Style Sheets
MIT	Massachusetts Institute of Technology
MVC	Model View Controller
ORM	Object Relational Mapping
QML	Qt MetaLanguage
RDBMS	Relational Database Management System
REST	Representational State Transfer
SASS	Syntactically Awesome Style Sheets
SGML	Standard Generalized Markup Language
SQL	Structured Query Language
URL	Uniform Resource Locator
W3C	WorldWideWeb Consortium
YAML	Yet Another Markup Language

8 Anexos

8.1 Definición del algoritmo húngaro

El algoritmo funciona empleando matrices de coste de dimensiones n×n. En el caso de que las filas y columnas fueran asimétricas, se pueden añadir entradas adicionales con coste 0, que podrán ser ignoradas tras realizar el algoritmo. Una vez construida la matriz de costes, se le aplicarán los siguientes pasos:

1. Para cada fila, encontrar el elemento más pequeño y sustraerlo de cada elemento de la fila.
2. Para cada columna, encontrar el elemento más pequeño y sustraerlo de cada elemento de la columna
3. Trazar el mínimo número de líneas horizontales o verticales para cubrir todos los valores nulos.
4. Si el número de líneas coincide con el tamaño de la matriz(n), la asignación de un número óptimo de ceros es posible y hemos terminado.
   Si por el contrario el número de líneas trazadas es inferior al tamaño de la matriz(n), procederemos al paso 5.
5. De todos los costes no cubiertos por un trazo, identificar el valor más pequeño. Sustraer dicho valor de todos los elementos de la matriz. Seguidamente sumar el valor a cada celda cada vez que esté cubierto por un trazo. Repetir desde el paso 3.

8.1.1 Ejemplo de funcionamiento [7]

Partimos de la matriz de costes anterior, de la cual identificamos para cada fila el menor valor:

	Juan	Paco	Ana	María
Interfaz	83	36	**34**	98
Algoritmo	42	75	39	**5**
Red	47	42	52	**34**
Sonido	**10**	35	28	72

Paso 1: Sustraemos en cada fila el valor más bajo de dicha fila.

	Juan	Paco	Ana	María
Interfaz **(-34)**	49	**2**	**0**	64
Algoritmo **(-5)**	37	70	34	**0**
Red **(-34)**	13	8	18	0
Sonido **(-10)**	**0**	25	18	62

Marcados en **azul** los valores mínimos de cada columna.
Paso 2: Sustraemos en cada columna el valor más bajo de dicha columna

	Juan (-0)	Paco (-2)	Ana (-0)	María (-0)
Interfaz	42	0	0	64
Algoritmo	37	68	34	0
Red	13	6	18	0
Sonido	0	23	18	62

Paso 3: Trazamos el mínimo número de líneas horizontales y verticales de tal modo que todos los ceros queden cubiertos.

	Juan	Paco	Ana	María
Interfaz	*42*	***0***	***0***	*64*
Algoritmo	37	68	34	***0***
Red	13	**6**	18	***0***
Sonido	***0***	*23*	*18*	*62*

Paso 4: El número de líneas trazadas para cubrir todos los ceros es 3, que es menor que el tamaño de las filas o columnas de la matriz (4). La asignación de un número óptimo de ceros no es aún posible por lo que procederemos al paso 5.

Paso 5: El valor más pequeño no cubierto es 6, que sustraemos de toda la matriz. El mismo valor se suma cada vez que una celda es cubierta por un trazo.

	Juan	Paco	Ana	María
Interfaz	42-6+6=42	0-6+6=0	0-6+6=0	64-6+6+6=70
Algoritmo	37-6=31	68-6=62	34-6=28	0-6+6=0
Red	13-6=7	6-6=0	18-6=12	0-6+6=0
Sonido	0-6+6=0	23-6+6=23	18-6+6=18	62-6+6+6=68

Paso 3: De nuevo trazamos el mínimo número de líneas horizontales y verticales de tal modo que todos los ceros queden cubiertos.

	Juan	Paco	Ana	María
Interfaz	42	**0**	**0**	70
Algoritmo	31	62	28	**0**
Red	7	**0**	12	**0**
Sonido	**0**	23	18	68

Paso 4: El número mínimo de líneas es 4, coincidiendo con el tamaño de la matriz. Por lo tanto hemos dado con el número óptimo de ceros. La solución queda marcada como las celdas en verde, escogidas como única asignación por descartes.

La asignación resultante es la siguiente:

	Juan	Paco	Ana	María
Interfaz	83	36	**34**	98
Algoritmo	42	75	39	**5**
Red	47	**42**	52	34
Sonido	**10**	35	28	72

El reparto final tiene un coste de 10+42+34+5=**91**

8.2 Nivel de aislamiento transaccional [14]

Transacción es el conjunto de instrucciones que se envían a la base de datos como un único bloque. ACID especifica 'I' como la necesidad de aislamiento entre transacciones. No obstante, un aislamiento estricto supone una pérdida considerable del rendimiento de la base de datos, por lo que los RDBMS suelen relajar por defecto el nivel de aislamiento, lo que puede ocasionar condiciones de carrera si el programador no tiene cuidado.

A continuación se enumeran los distintos grados de aislamiento definidos por el estándar SQL, así como las distintas lecturas erróneas a las que son vulnerables. Los ejemplos muestran la interacción entre dos transacciones distintas, A y B. Supongamos que la tabla T tiene una única fila con la columna x=1.

8.2.1 Read uncommitted:

Es el grado más bajo de aislamiento (ninguno). Por lo general, ninguna base de datos utiliza este grado de aislamiento por defecto. Es vulnerable a **dirty reads**.

Dirty reads:

Ocurre cuando se lee el valor de una tabla antes de haberse realizado commit. El valor puede ser inválido al tratarse de una transacción que nunca se materializa.

```
A: select x from t;        -- x = 1

B: update t set x=10;

A: select x from t;        -- x = 10

B: rollback;               -- Update no se realiza, x=1
```

8.2.2 Read committed:

Como protección a *Dirty reads* bloquea las filas afectadas por update hasta que no se realiza commit. Es vulnerable a **non-repeatable reads**.

Non-repeatable reads:

Ocurre también cuando se lee el valor de una tabla antes de haberse realizado commit. Aunque el valor sea válido, **A** lee valores distintos en una misma transacción cuando debería leer el mismo.

```
A: select x from t;        -- x = 1

B: update t set x=10;

B: commit;

A: select x from t;        -- x = 10

A: commit;
```

8.2.3 Repeatable reads:

En este caso el bloqueo no únicamente se mantiene hasta que termine la transacción que modifica la tabla, sino también hasta que acabe la transacción lectora. Es vulnerable a **Phantom reads**.

Phantom reads:

Mantener cerrojos a nivel de fila hasta que ambas transacciones se hayan materializado no es suficiente, ya que la transacción escritora podría insertar una fila nueva.

```
A: select x from t;          -- x = 1

B: insert into t values(2)

B: commit;

A: select x from t;          -- x = 1, x = 2

A: commit;
```

8.2.4 Serializable:

Es el nivel más alto de aislamiento, y también el más costoso, ya que requiere emplear cerrojos de escritura y lectura para las filas implicadas, así como cerrojos adicionales para controlar el número de filas para evitar la situación de **phantom reads**.
Existe una variedad de este tipo de aislamiento conocido como **snapshot isolation**, que se fundamenta en tomar capturas de la tabla al principio de la transacción para posteriormente comparar y comprobar que no ha ocurrido una escritura sobre escritura (en tal caso, la transacción se abortará). Esto produce una mejora considerable de rendimiento al no requerir emplear cerrojos, pero no queda exento de que puedan ocurrir otros casos más raros de condiciones de carrera.

8.2.5 Nivel de aislamiento transaccional escogido para el proyecto:

Para operaciones largas como el intercambio de tareas habrá que realizar múltiples comprobaciones y múltiples actualizaciones dentro de una transacción. Una condición de carrera en la cual se acepten dos intercambios a la vez podría resultar desastroso, ya que podría incurrir en el rechazo de intercambios que no se viesen involucrados y en una re-asignación caótica de las tareas que podrían llegar a violar incluso las restricciones fuertes.

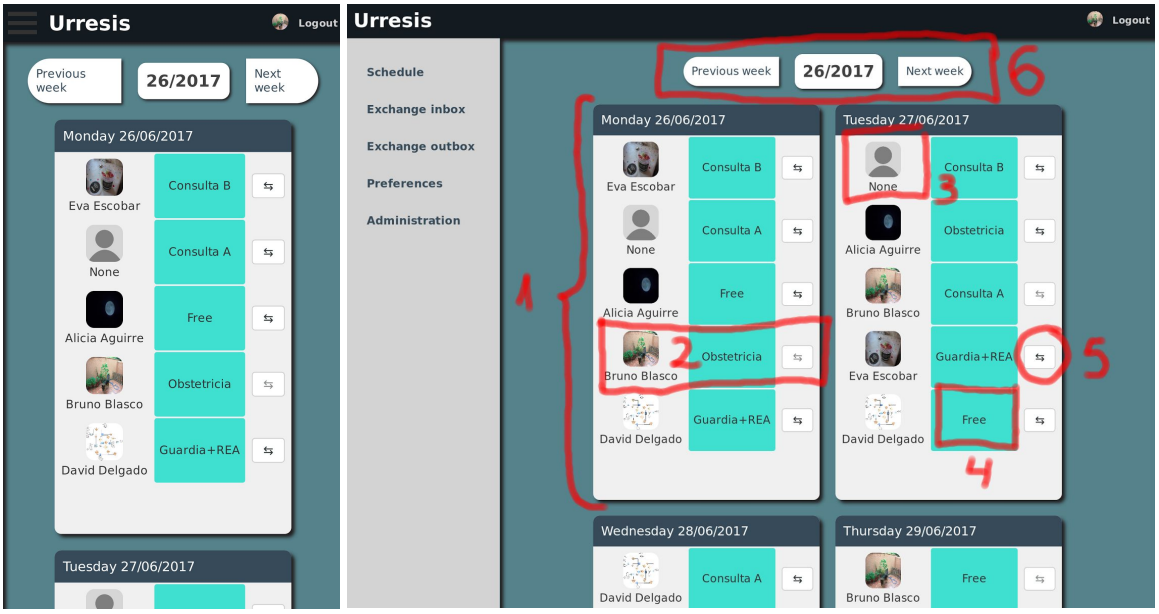
Por otra parte, y teniendo en cuenta que sería extraño que un hospital tuviese una plantilla con más que cientos de anestesistas, el tráfico que realizase cambios en la base de datos es muy limitado y puntual, siendo además insignificante la repercusión de rendimiento en comparativa con las propias operaciones.

Por estos motivos se ha decidido el máximo nivel de aislamiento: **Serializable**.

8.3 Aspecto de las distintas secciones de la plataforma

8.3.1 Calendario con planificación

Esta sección es la parte más importante de la página. Desde aquí se puede ver el reparto de las tareas a nivel semanal. Tras realizar cualquier operación como la de actualizar los datos del perfil, se volverá de nuevo a esta página.



1. **Ventana**: Contiene la planificación de un día de la semana. En el título se encuentra el día y en el cuerpo el reparto.

2. **Tarea**: Cada fila consiste en la asignación de un trabajo a un trabajador.

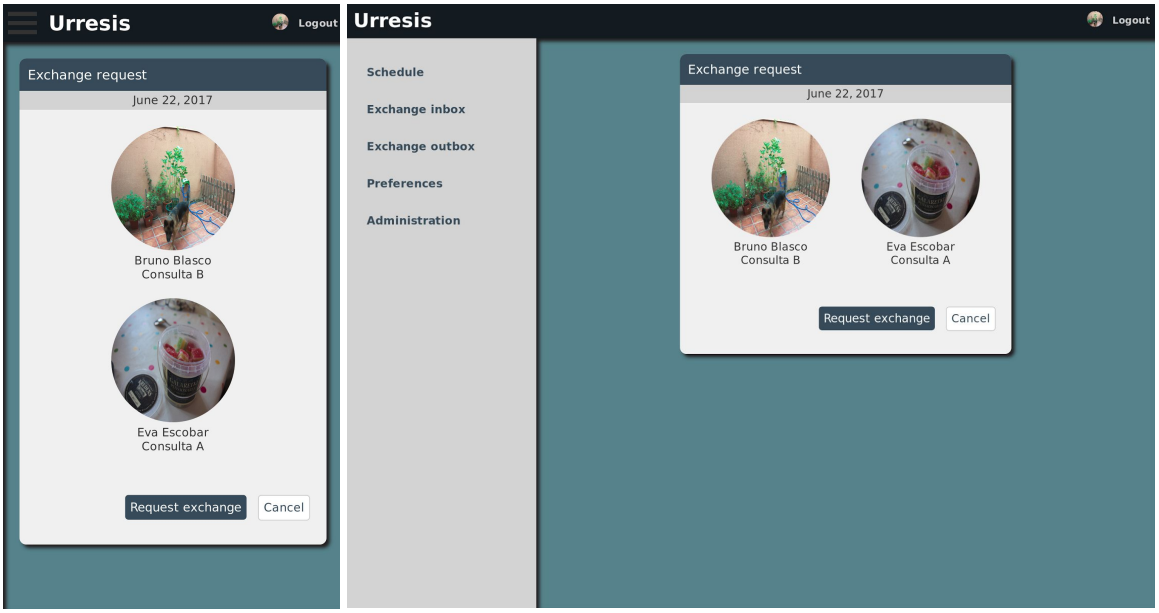
3. **Trabajador**: Nombre del trabajador al que se le ha asignado una tarea. Encima se muestra su foto de perfil.
   En este caso no hay suficientes trabajadores disponibles a repartir entre todos los trabajos, por lo que el trabajador asignado a la consulta B (de prioridad pequeña) se denota como None.

4. **Trabajo**: La columna turquesa son los trabajos asignados.
   En este caso en concreto se puede apreciar que David Delgado está libre. Esto se debe a que ha trabajado el día anterior en una guardia. Si escaseasen las tareas, más trabajadores aparecerían como libres.

5. **Botón de solicitud de intercambio**: Cuando a un trabajador le interesa un trabajo, pincha sobre este botón para intercambiar dicha tarea con la suya propia del mismo día. El botón está desactivado para intercambiar consigo mismo y oculto si el usuario no tiene un trabajador asociado.

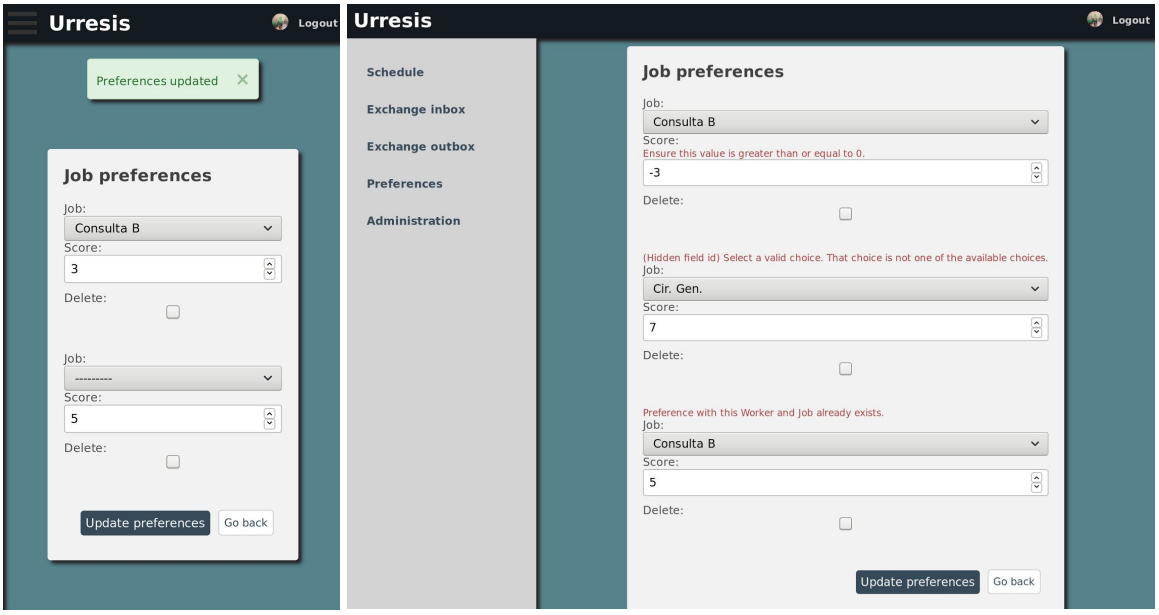
6. **Barra de navegación del calendario**: Mediante estas entradas puede cambiarse a ver el plan de otra semana.

8.3.2 Ventana de solicitud del intercambio



Cuando el usuario está solicitando un intercambio, puede emplear este diálogo para confirmar la solicitud. Se muestra como subtítulo de ventana el día y en el cuerpo las tareas implicadas. En la base botones para confirmar o cancelar.

8.3.3 Preferencias

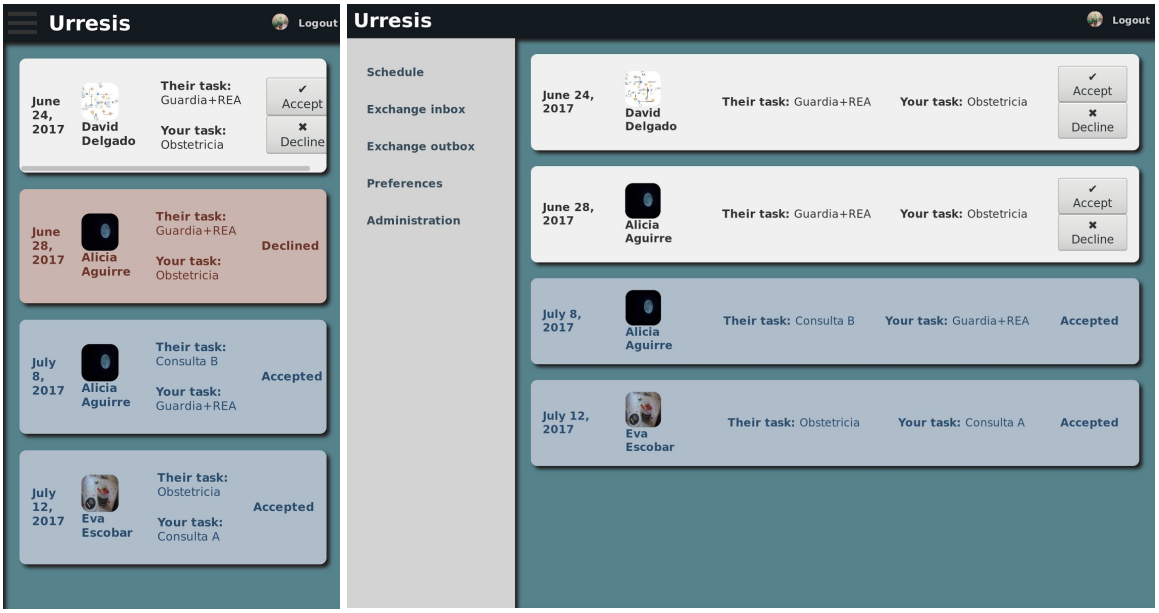


Desde esta sección el trabajador puede especificar sus preferencias a realizar determinadas tareas. A mayor el valor, más preferible la tarea. La última entrada se utiliza para introducir preferencias nuevas.

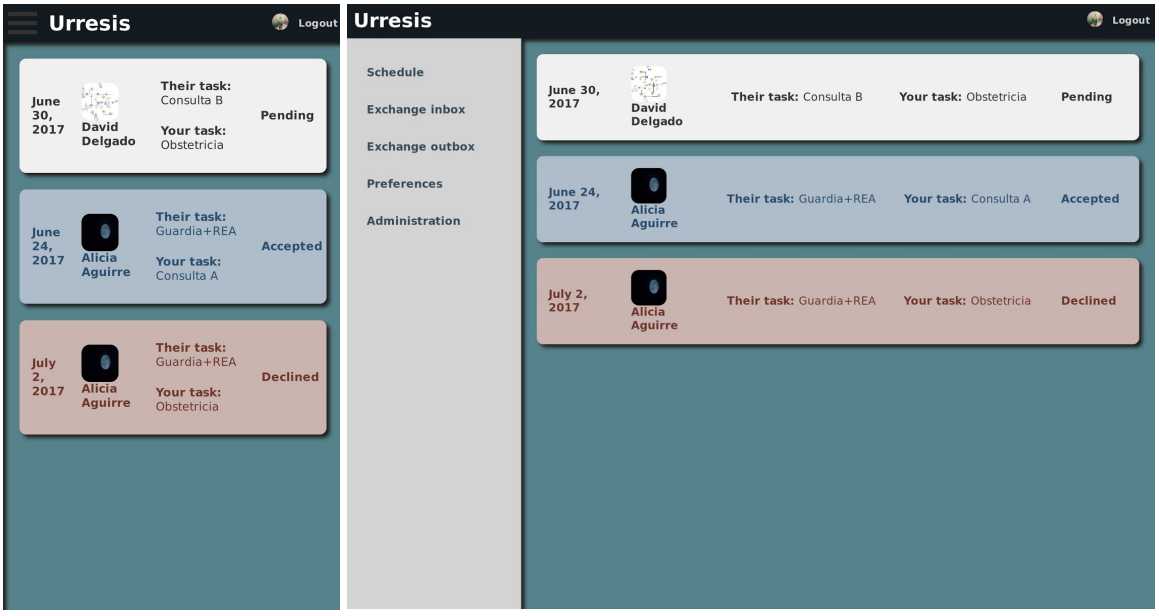
Esta sección está oculta a usuarios que no tengan trabajadores asociados.

8.3.4 Listado de intercambios

8.3.4.1 Buzón de entrada



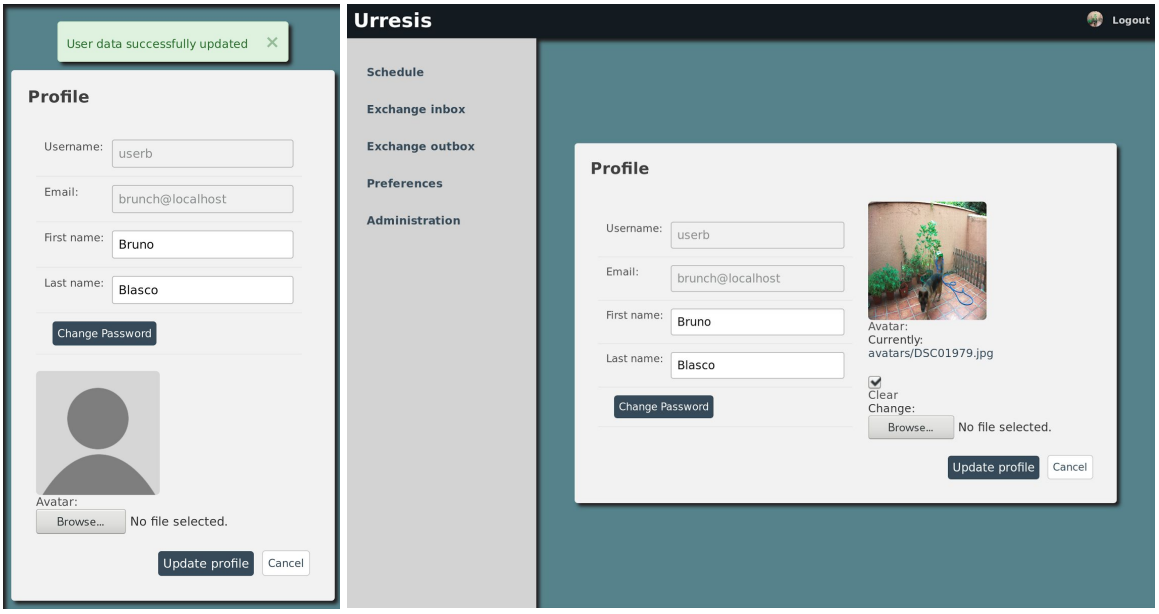
8.3.4.2 Buzón de salida



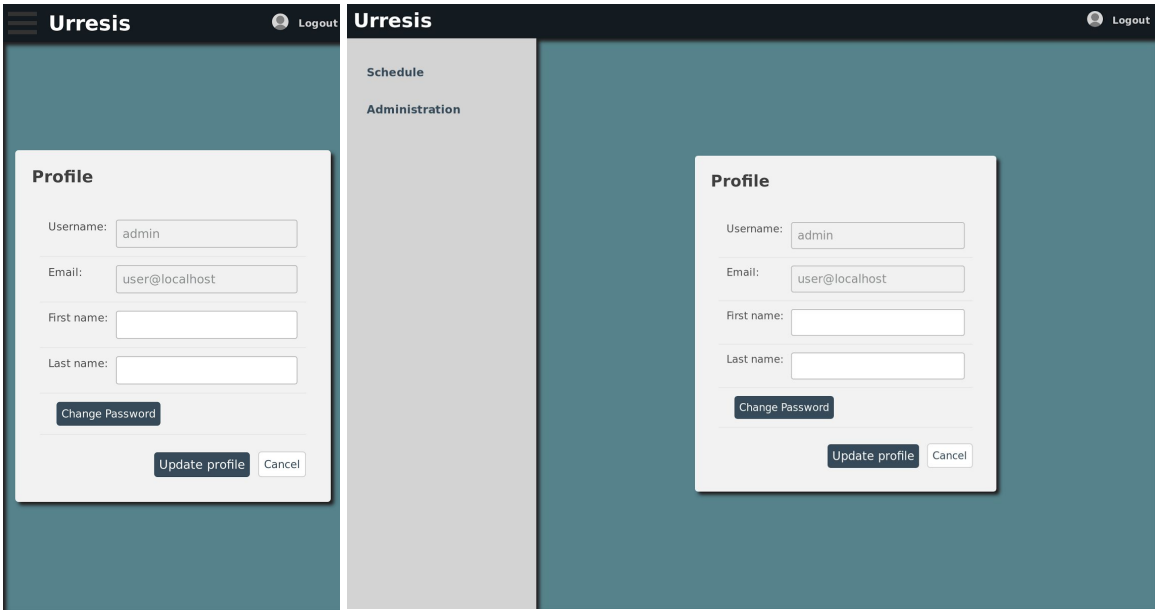
Desde estas secciones el trabajador puede gestionar las solicitudes de intercambio que reciben y controlar el estado de las solicitudes propias.
Las solicitudes están ordenadas por temporalmente, las más próximas primero. Las solicitudes pendientes siempre se muestran antes.

Estas secciones están ocultas a los usuarios que no tengan un trabajador asociado

8.3.5 Perfil

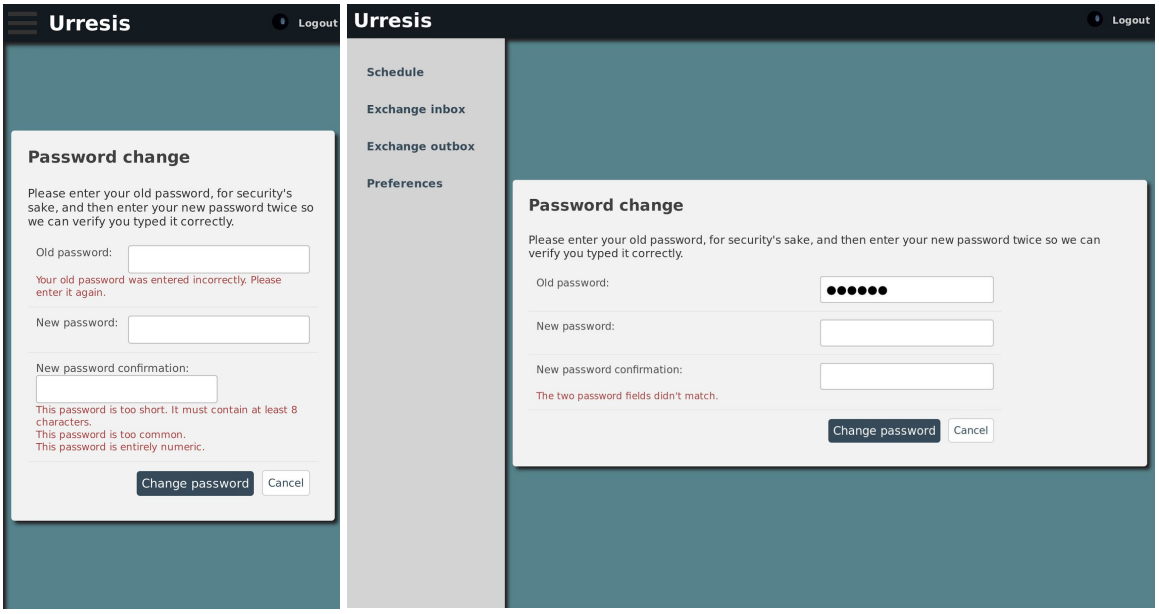


Desde aquí el usuario puede modificar los detalles de su perfil y acceder al cambio de contraseña. Tanto el nombre como el avatar se mostrarán en la planificación semanal.



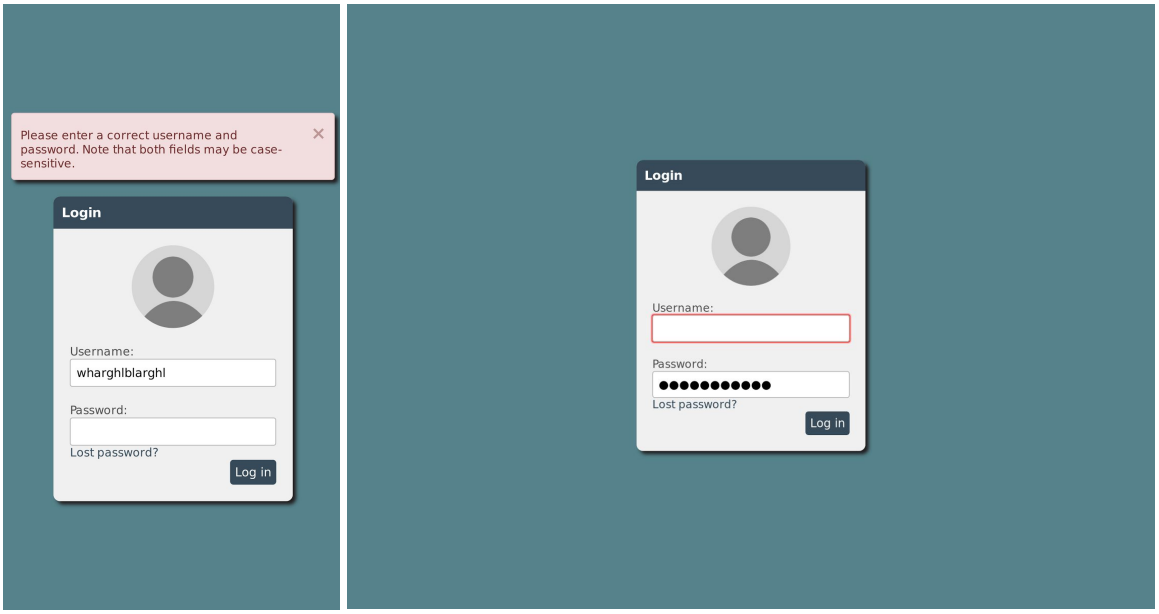
En caso de que el usuario no tenga un trabajador asociado, se ocultarán las secciones del trabajador

8.3.6 Cambio de contraseña



Desde el perfil el usuario puede acceder a este menú para cambiar de contraseña.
Si el usuario formula una contraseña insegura o se equivoca, aparecerán pequeños diálogos en rojo en las entradas erróneas indicando el problema.

8.3.7 Página de login

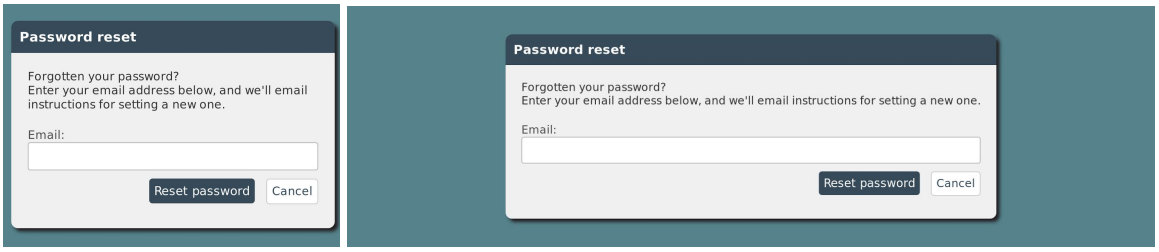


La página entera está protegida mediante login. Si el usuario olvida su contraseña, puede restaurarla pinchando en 'lost password'. En caso de equivocación, se muestra un mensaje de error y la ventana hace una animación de sacudida.

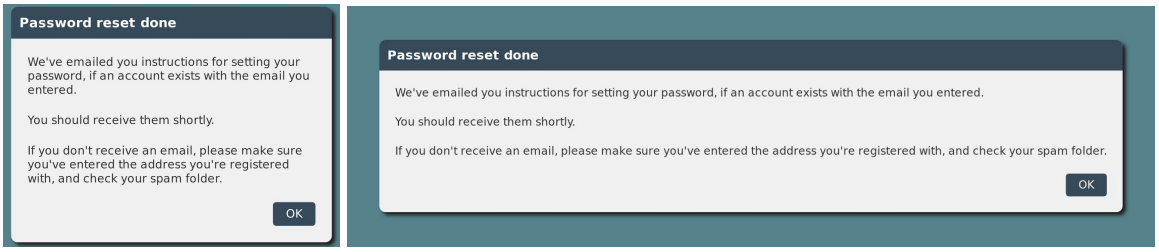
8.3.8 Restauración de contraseña

Si el usuario olvida su contraseña, puede introducir su email, donde se le enviará un correo electrónico con un enlace en el que viene incrustado un token para solicitar la restauración de la contraseña.
Siguiendo ese enlace podrá introducir una nueva contraseña.

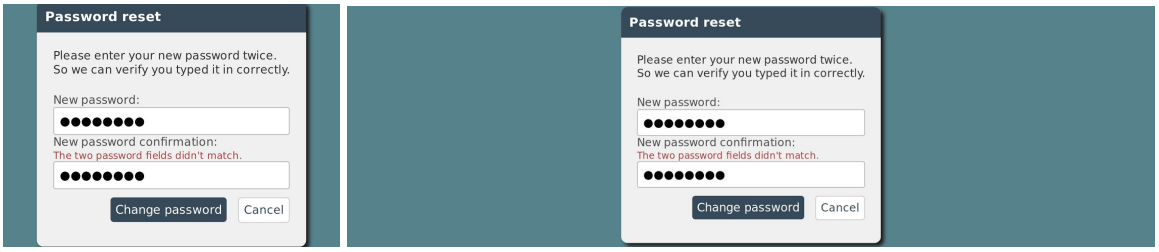
8.3.8.1 Introducción de email



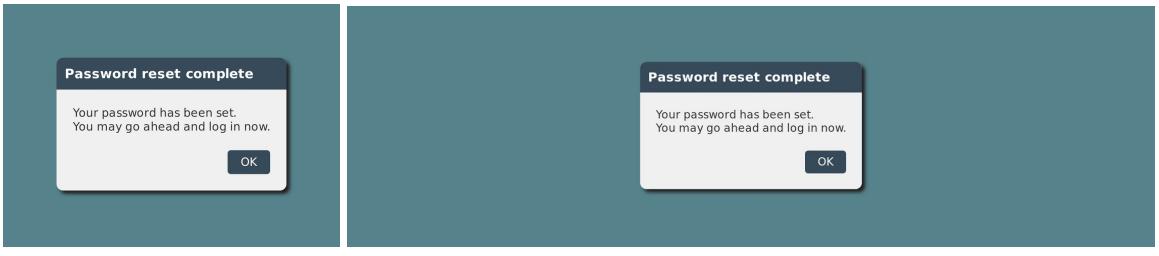
8.3.8.2 Confirmación de envío



8.3.8.3 Nueva contraseña

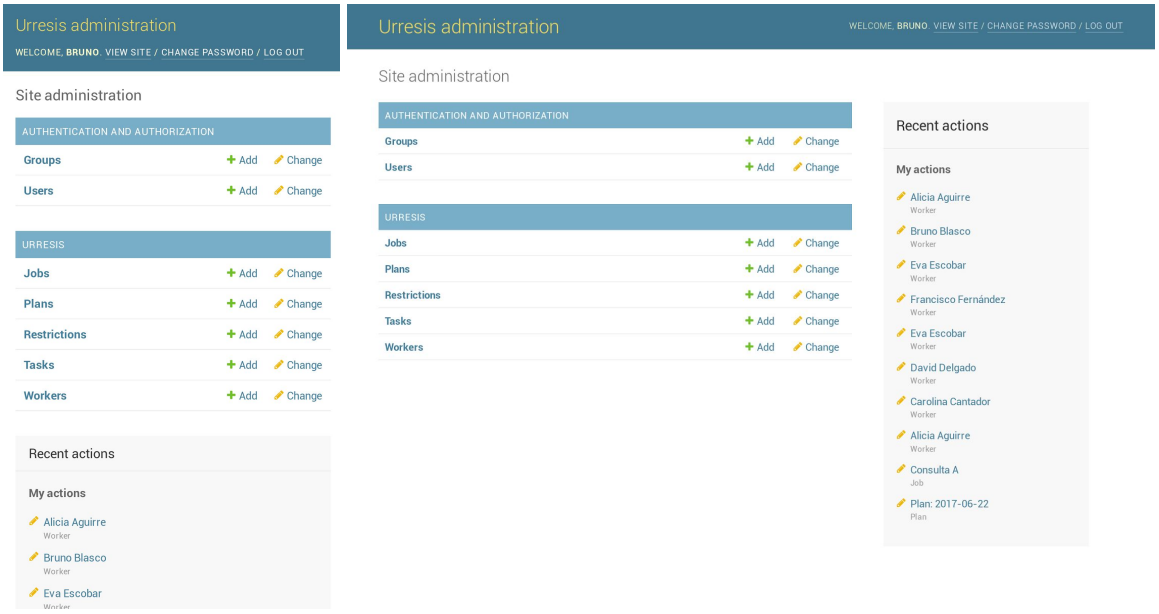


8.3.8.4 Nueva contraseña



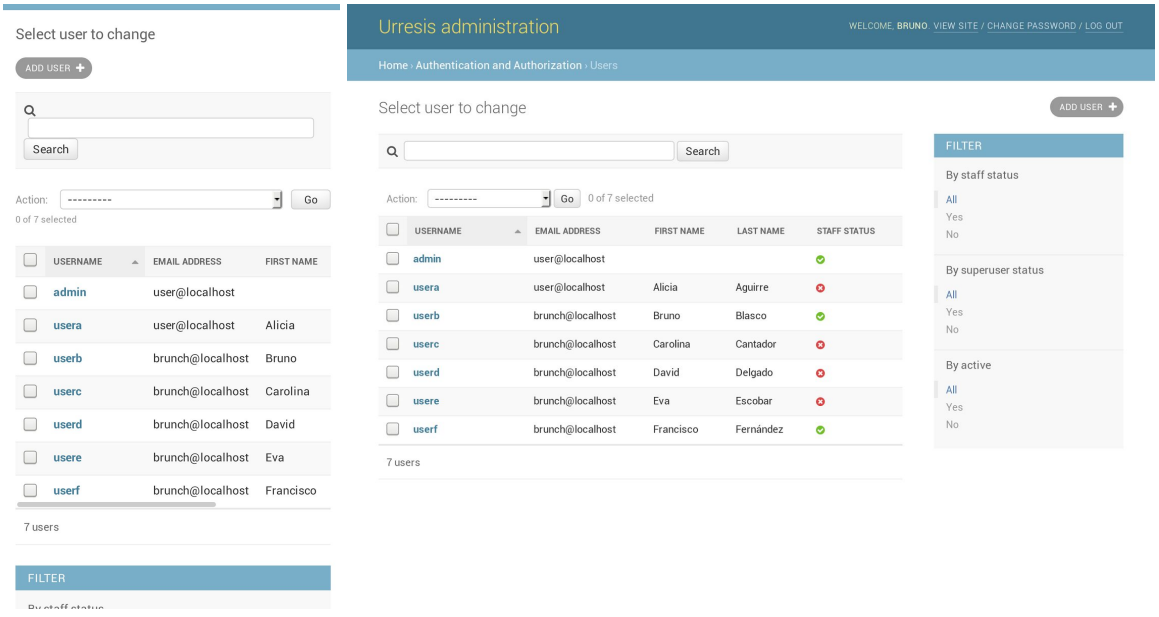
8.4 Aspecto de la sección de administración

8.4.1 Sección principal



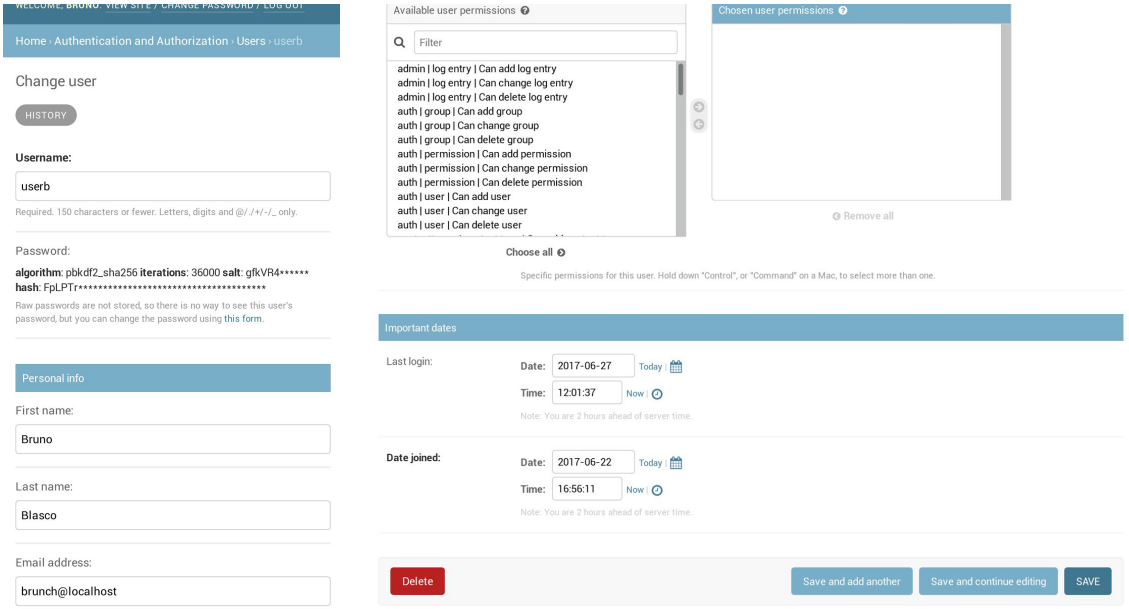
Esta sección está separada entre los apartados relacionados a la autenticación y usuarios y los apartados relacionados con la aplicación en sí. A la derecha se muestra un bloque con las acciones recientes realizadas en administración.

8.4.2 Listado de entidad



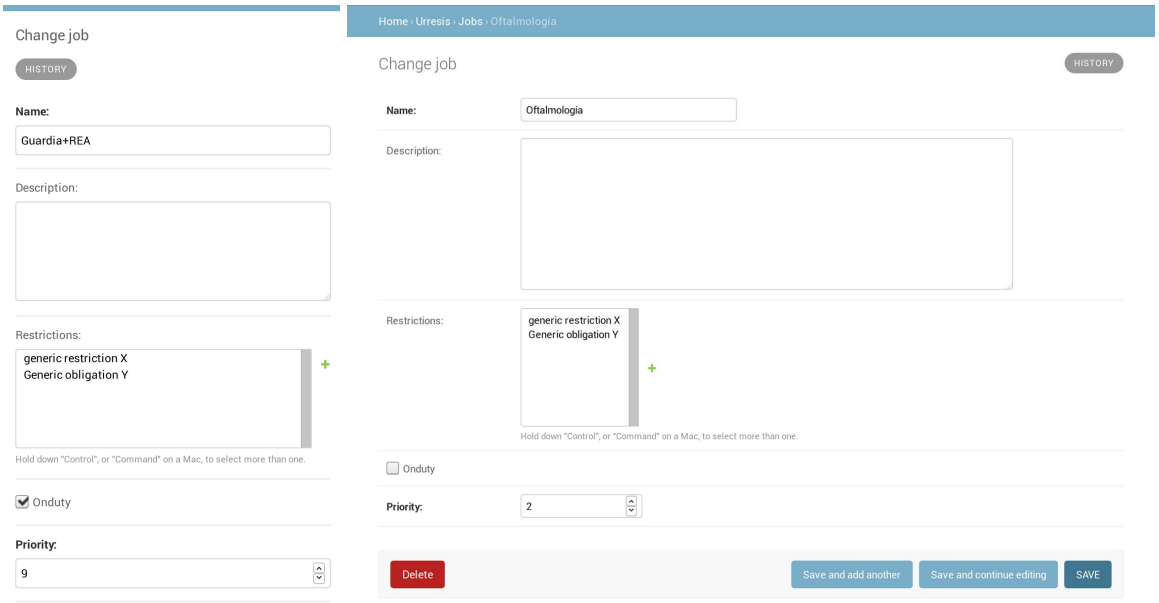
Al pinchar en cualquiera de los elementos desde el menú principal, se despliega una lista con instancias almacenadas en la base de datos de una determinada entidad. En este caso, se muestran los distintos usuarios con información resumida de sus atributos, pero existe una página similar para cada una de las entidades registradas.

8.4.3 Detalle de usuario



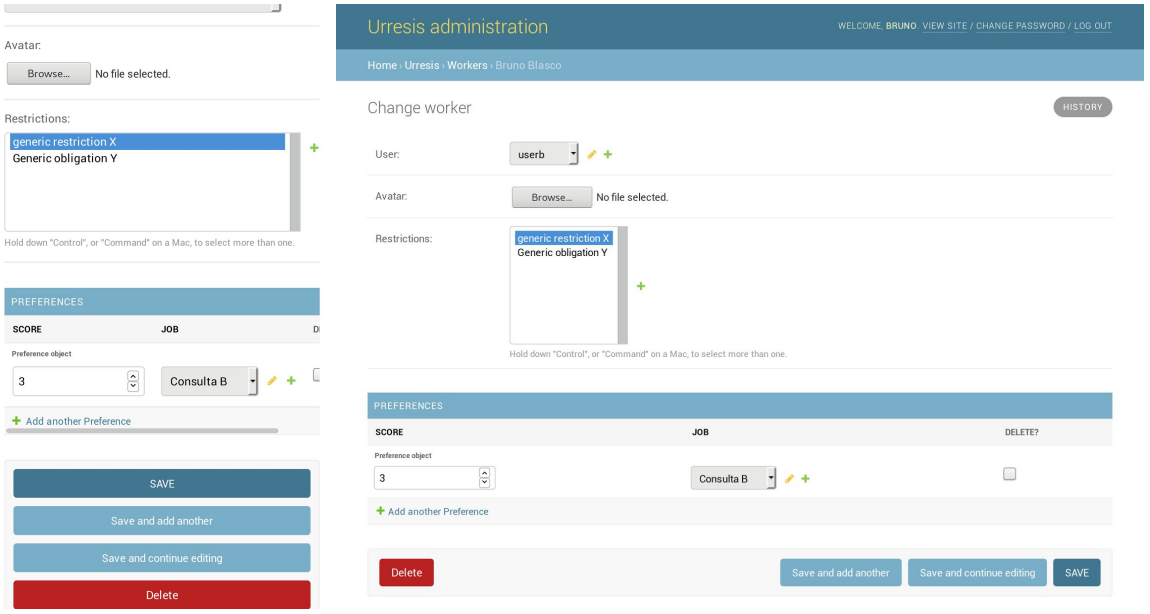
El administrador puede desde aquí modificar datos relevantes a los usuarios, como su nombre de usuario, permisos, marcarlo como administrador (parcial o total) e incluso modificar su contraseña.

8.4.4 Detalle de trabajo



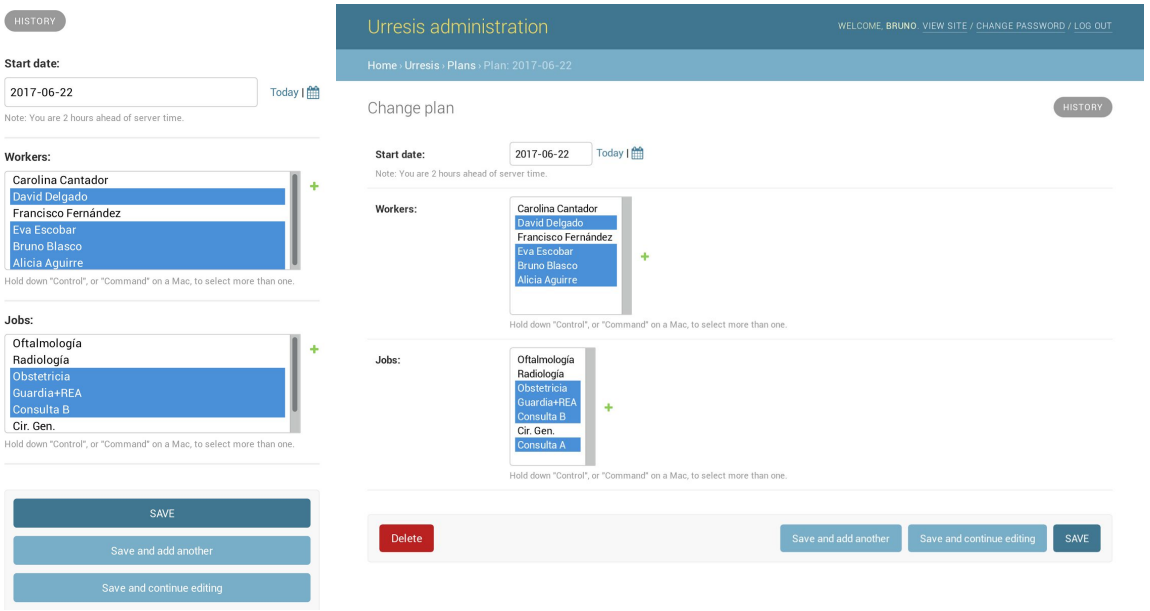
A los trabajos se les puede añadir una pequeña descripción, las restricciones asociadas (que por comodidad también se pueden crear nuevas rellenando un *popup* al clicar sobre la cruz verde), si es guardia o no y la prioridad del trabajo.

8.4.5 Detalle de trabajador



Del trabajador se puede modificar el usuario que tiene asociado (que puede ser ninguno), su avatar, restricciones y preferencias.

8.4.6 Detalle plan



Para el plan se puede seleccionar el día sobre el que comienza utilizando un pequeño calendario.
Adicionalmente se deben seleccionar los trabajos y trabajadores implicados en dicho plan.